TRAITE DE COOPERATION EN MATIERE DE BREVETS

Expéditeur : le BUREAU INTERNATIONAL

PCT

NOTIFICATION RELATIVE A LA PRESENTATION OU A LA TRANSMISSION DU DOCUMENT DE PRIORITE

(instruction administrative 411 du PCT)

Destinataire:

CORLU, Bernard
Bull S.A.
PC 62A24
68, route de Versailles
Boîte postale 45
F-78434 Louveciennes Cedex
FRANCE

Date d'expédition (jour/mois/année) 26 juin 2001 (26.06.01)	
Référence du dossier du déposant ou du mandataire PCT 3884/BC	**NOTIFICATION IMPORTANTE**
Demande internationale no PCT/FR01/01506	Date du dépôt international (jour/mois/année) 17 mai 2001 (17.05.01)
Date de publication internationale (jour/mois/année) Pas encore publiée	Date de priorité (jour/mois/année) 17 mai 2000 (17.05.00)
Déposant BULL CP8 etc	

1. La date de réception (sauf lorsque les lettres "NR" figurent dans la colonne de droite) par le Bureau international du ou des documents de priorité correspondant à la ou aux demandes énumérées ci-après est notifiée au déposant. Sauf indication contraire consistant en un astérisque figurant à côté d'une date de réception, ou les lettres "NR", dans la colonne de droite, le document de priorité en question a été présenté ou transmis au Bureau international d'une manière conforme à la règle 17.1.a) ou b).
2. Ce formulaire met à jour et remplace toute notification relative à la présentation ou à la transmission du document de priorité qui a été envoyée précédemment.
3. Un **astérisque(*)** figurant à côté d'une date de réception dans la colonne de droite signale un document de priorité présenté ou transmis au Bureau international mais de manière non conforme à la règle 17.1.a) ou b). Dans ce cas, **l'attention du déposant est appelée** sur la règle 17.1.c) qui stipule qu'aucun office désigné ne peut décider de ne pas tenir compte de la revendication de priorité avant d'avoir donné au déposant la possibilité de remettre le document de priorité dans un délai raisonnable en l'espèce.
4. Les **lettres "NR"** figurant dans la colonne de droite signalent un document de priorité que le Bureau international n'a pas reçu ou que le déposant n'a pas demandé à l'office récepteur de préparer et de transmettre au Bureau international, conformément à la règle 17.1.a) ou b), respectivement. Dans ce cas, **l'attention du déposant est appelée** sur la règle 17.1.c) qui stipule qu'aucun office désigné ne peut décider de ne pas tenir compte de la revendication de priorité avant d'avoir donné au déposant la possibilité de remettre le document de priorité dans un délai raisonnable en l'espèce.

Date de priorité	Demande de priorité n°	Pays, office régional ou office récepteur selon le PCT	Date de réception du document de priorité
17 mai 2000 (17.05.00)	00/06882	FR	19 juin 2001 (19.06.01)

Bureau international de l'OMPI **34, chemin des Colombettes** **1211 Genève 20, Suisse** no de télécopieur (41-22) 740.14.35	Fonctionnaire autorisé: P. Blanchet (Fax 338.87.40) no de téléphone (41-22) 338.83.38

This Page Blank (uspto)

INPI
INSTITUT NATIONAL DE LA PROPRIETE INDUSTRIELLE

BREVET D'INVENTION
CERTIFICAT D'UTILITÉ

REQUÊTE EN DÉLIVRANCE 2/2

Réservé à l'INPI

REMISE DES PIÈCES
DATE 17 MAI 2000
LIEU 75 INPI PARIS F
N° D'ENREGISTREMENT
NATIONAL ATTRIBUÉ PAR L'INPI 0006882

DB 540 W /260899

Vos références pour ce dossier : *(facultatif)*		FR 3884/BC
6 MANDATAIRE		
Nom		CORLU
Prénom		Bernard
Cabinet ou Société		BULL S.A.
N °de pouvoir permanent et/ou de lien contractuel		PG 4280
Adresse	Rue	68, route de Versailles / PC 58D20
	Code postal et ville	78434 LOUVECIENNES CEDEX
N° de téléphone *(facultatif)*		01.39.66.61.76
N° de télécopie *(facultatif)*		01.39.66.61.73
Adresse électronique *(facultatif)*		BERNARD.CORLU@BULL.NET
7 INVENTEUR (S)		
Les inventeurs sont les demandeurs		☐ Oui ☒ Non **Dans ce cas fournir une désignation d'inventeur(s) séparée**
8 RAPPORT DE RECHERCHE		**Uniquement pour une demande de brevet (y compris division et transformation)**
Établissement immédiat ou établissement différé		☒ ☐
Paiement échelonné de la redevance		**Paiement en trois versements, uniquement pour les personnes physiques** ☐ Oui ☐ Non
9 RÉDUCTION DU TAUX DES REDEVANCES		**Uniquement pour les personnes physiques** ☐ Requise pour la première fois pour cette invention *(joindre un avis de non-imposition)* ☐ Requise antérieurement à ce dépôt *(joindre une copie de la décision d'admission pour cette invention ou indiquer sa référence)*:
Si vous avez utilisé l'imprimé «Suite», indiquez le nombre de pages jointes		0

10 SIGNATURE DU DEMANDEUR OU DU MANDATAIRE (Nom et qualité du signataire) CORLU Bernard (Mandataire)	**VISA DE LA PRÉFECTURE OU DE L'INPI**

La loi n°78-17 du 6 janvier 1978 relative à l'informatique, aux fichiers et aux libertés s'applique aux réponses faites à ce formulaire. Elle garantit un droit d'accès et de rectification pour les données vous concernant auprès de l'INPI.

This Page Blank (uspto)

INPI
INSTITUT NATIONAL DE LA PROPRIETE INDUSTRIELLE

BREVET D'INVENTION
CERTIFICAT D'UTILITÉ

Code de la propriété intellectuelle - Livre VI

cerfa
N° 11235*02

DÉPARTEMENT DES BREVETS

26 bis, rue de Saint Pétersbourg
75800 Paris Cedex 08
Téléphone : 01 53 04 53 04 Télécopie : 01 42 94 86 54

DÉSIGNATION D'INVENTEUR(S) Page N° 1 / 1

(Si le demandeur n'est pas l'inventeur ou l'unique inventeur)

Cet imprimé est à remplir lisiblement à l'encre noire

DB 113 W /260899

Vos références pour ce dossier *(facultatif)*	FR 3884/BC
N° D'ENREGISTREMENT NATIONAL	

TITRE DE L'INVENTION (200 caractères ou espaces maximum)

Procédé de sécurisation d'un langage du type à données typées, notamment dans un système embarqué et système embarqué de mise en œuvre du procédé.

LE(S) DEMANDEUR(S) :

BULL CP8
BP 45 - 68, route de Versailles
78430 LOUVECIENNES - FRANCE

DESIGNE(NT) EN TANT QU'INVENTEUR(S) : (Indiquez en haut à droite «Page N° 1/1» S'il y a plus de trois inventeurs, utilisez un formulaire identique et numérotez chaque page en indiquant le nombre total de pages).

Nom		Fougeroux	
Prénoms		Nicolas	
Adresse	Rue	6 square Bainville	
	Code postal et ville	78150	LE CHESNAY - FRANCE
Société d'appartenance *(facultatif)*			
Nom		Hameau	
Prénoms		Patrice	
Adresse	Rue	18 rue Belle Feuille	
	Code postal et ville	92100	BOULOGNE BILLANCOURT - FRANCE
Société d'appartenance *(facultatif)*			
Nom		Landier	
Prénoms		Olivier	
Adresse	Rue	138 avenue Felix Faure	
	Code postal et ville	75015	PARIS - FRANCE
Société d'appartenance *(facultatif)*			

DATE ET SIGNATURE(S) DU (DES) DEMANDEUR(S) OU DU MANDATAIRE (Nom et qualité du signataire)	Louveciennes, le 29 mai 2000 CORLU Bernard (Mandataire)

La loi n°78-17 du 6 janvier 1978 relative à l'informatique, aux fichiers et aux libertés s'applique aux réponses faites à ce formulaire. Elle garantit un droit d'accès et de rectification pour les données vous concernant auprès de l'INPI.

This Page Blank (uspto)

PCT

REQUÊTE

Le soussigné requiert que la présente demande internationale soit traitée conformément au Traité de coopération en matière de brevets.

Réservé à l'office récepteur
Demande internationale n°
Date du dépôt international
Nom de l'office récepteur et "Demande internationale PCT"

Référence du dossier du déposant ou du mandataire *(facultatif)* *(12 caractères au maximum)* PCT 3884/BC

Cadre n° I TITRE DE L'INVENTION **Procédé de sécurisation d'un langage du type à données typées, notamment dans un système embarqué et système embarqué de mise en œuvre du procédé.**

Cadre n° II DÉPOSANT

Nom et adresse : *(Nom de famille suivi du prénom; pour une personne morale, désignation officielle complète. L'adresse doit comprendre le code postal et le nom du pays. Le pays de l'adresse indiquée dans ce cadre est l'État où le déposant a son domicile si aucun domicile n'est indiqué ci-dessous.)*

BULL CP8
68, route de Versailles
BP 45
78430 LOUVECIENNES
FRANCE

☐ Cette personne est aussi inventeur.

n° de téléphone (33) 1 39.66.61.76

n° de télécopieur (33) 1 39.66.43.36

n° de téléimprimeur

Nationalité (nom de l'État) : FRANCE

Domicile (nom de l'État) : FRANCE

Cette personne est déposant pour : ☐ tous les États désignés ☒ tous les États désignés sauf les États-Unis d'Amérique ☐ les États-Unis d'Amérique seulement ☐ les États indiqués dans le cadre supplémentaire

Cadre n° III AUTRE(S) DÉPOSANT(S) OU (AUTRE(S)) INVENTEUR(S)

Nom et adresse : *(Nom de famille suivi du prénom; pour une personne morale, désignation officielle complète. L'adresse doit comprendre le code postal et le nom du pays. Le pays de l'adresse indiquée dans ce cadre est l'État où le déposant a son domicile si aucun domicile n'est indiqué ci-dessous.)*

Fougeroux Nicolas
6 square Bainville
78150 LE CHESNAY
FRANCE

Cette personne est :

☐ déposant seulement

☒ déposant et inventeur

☐ inventeur seulement *(Si cette case est cochée, ne pas remplir la suite.)*

Nationalité (nom de l'État) : FRANCE

Domicile (nom de l'État) : FRANCE

Cette personne est déposant pour : ☐ tous les États désignés ☐ tous les États désignés sauf les États-Unis d'Amérique ☒ les États-Unis d'Amérique seulement ☐ les États indiqués dans le cadre supplémentaire

☒ D'autres déposants ou inventeurs sont indiqués sur une feuille annexe.

Cadre n° IV MANDATAIRE OU REPRÉSENTANT COMMUN; OU ADRESSE POUR LA CORRESPONDANCE

La personne dont l'identité est donnée ci-dessous est/a été désignée pour agir au nom du ou des déposants auprès des autorités internationales compétentes, comme: ☒ mandataire ☐ représentant commun

Nom et adresse : *(Nom de famille suivi du prénom; pour une personne morale, désignation officielle complète. L'adresse doit comprendre le code postal et le nom du pays.)*

BULL S.A
CORLU Bernard
PC 62A24 / 68, route de Versailles -BP45
F- 78434 LOUVECIENNES Cedex (FRANCE)

n° de téléphone (33) 1 39.66.61.76

n° de télécopieur (33) 1 39.66.43.36

n° de téléimprimeur

☐ **Adresse pour la correspondance :** cocher cette case lorsque aucun mandataire ni représentant commun n'est/n'a été désigné et que l'espace ci-dessus est utilisé pour indiquer une adresse spéciale à laquelle la correspondance doit être envoyée.

This Page Blank (uspto)

Feuille n° ....

Suite du cadre n° III AUTRE(S) DÉPOSANT(S) OU (AUTRE(S)) INVENTEUR(S)

Si aucun des sous-cadres suivants n'est utilisé, cette feuille ne doit pas être incluse dans la requête.

Nom et adresse : *(Nom de famille suivi du prénom; pour une personne morale, désignation officielle complète. L'adresse doit comprendre le code postal et le nom du pays. Le pays de l'adresse indiquée dans ce cadre est l'État où le déposant a son domicile si aucun domicile n'est indiqué ci-dessous.)*

Landier Olivier
138 avenue Félix Faure
75015 PARIS
FRANCE

Cette personne est :

- ☐ déposant seulement
- ☒ déposant et inventeur
- ☐ inventeur seulement *(Si cette case est cochée, ne pas remplir la suite.)*

Nationalité (nom de l'État) FRANCE

Domicile (nom de l'État) : FRANCE

Cette personne est déposant pour : ☐ tous les États désignés ☐ tous les États désignés sauf les États-Unis d'Amérique ☒ les États-Unis d'Amérique seulement ☐ les États indiqués dans le cadre supplémentaire

Nom et adresse : *(Nom de famille suivi du prénom; pour une personne morale, désignation officielle complète. L'adresse doit comprendre le code postal et le nom du pays. Le pays de l'adresse indiquée dans ce cadre est l'État où le déposant a son domicile si aucun domicile n'est indiqué ci-dessous.)*

Hameau Patrice
18 rue Belle Feuille
92100 BOULOGNE BILLANCOURT
FRANCE

Cette personne est :

- ☐ déposant seulement
- ☒ déposant et inventeur
- ☐ inventeur seulement *(Si cette case est cochée, ne pas remplir la suite.)*

Nationalité (nom de l'État) : FRANCE

Domicile (nom de l'État) : FRANCE

Cette personne est déposant pour : ☐ tous les États désignés ☐ tous les États désignés sauf les États-Unis d'Amérique ☒ les États-Unis d'Amérique seulement ☐ les États indiqués dans le cadre supplémentaire

Nom et adresse : *(Nom de famille suivi du prénom; pour une personne morale, désignation officielle complète. L'adresse doit comprendre le code postal et le nom du pays. Le pays de l'adresse indiquée dans ce cadre est l'État où le déposant a son domicile si aucun domicile n'est indiqué ci-dessous.)*

Cette personne est :

- ☐ déposant seulement
- ☐ déposant et inventeur
- ☐ inventeur seulement *(Si cette case est cochée, ne pas remplir la suite.)*

Nationalité (nom de l'État) :

Domicile (nom de l'État) :

Cette personne est déposant pour : ☐ tous les États désignés ☐ tous les États désignés sauf les États-Unis d'Amérique ☐ les États-Unis d'Amérique seulement ☐ les États indiqués dans le cadre supplémentaire

Nom et adresse : *(Nom de famille suivi du prénom; pour une personne morale, désignation officielle complète. L'adresse doit comprendre le code postal et le nom du pays. Le pays de l'adresse indiquée dans ce cadre est l'État où le déposant a son domicile si aucun domicile n'est indiqué ci-dessous.)*

Cette personne est :

- ☐ déposant seulement
- ☐ déposant et inventeur
- ☐ inventeur seulement *(Si cette case est cochée, ne pas remplir la suite.)*

Nationalité (nom de l'État) :

Domicile (nom de l'État) :

Cette personne est déposant pour : ☐ tous les États désignés ☐ tous les États désignés sauf les États-Unis d'Amérique ☐ les États-Unis d'Amérique seulement ☐ les États indiqués dans le cadre supplémentaire

☐ **D'autres déposants ou inventeurs sont indiqués sur une autre feuille annexe.**

This Page Blank (uspto)

Cadre n° V DÉSIGNATION D'ÉTATS

Les désignations suivantes sont fait[illegible]formément à la règle 4.9.a) *(cocher les cases app[illegible]es; une au moins doit l'être)* :

Brevet régional

- [ ] **AP** **Brevet ARIPO : GH** Ghana, **GM** Gambie, **KE** Kenya, **LS** Lesotho, **MW** Malawi, **MZ** Mozambique, **SD** Soudan, **SL** Sierra Leone, **SZ** Swaziland, **TZ** République-Unie de Tanzanie, **UG** Ouganda, **ZW** Zimbabwe et tout autre État qui est un État contractant du Protocole de Harare et du PCT
- [ ] **EA** **Brevet eurasien : AM** Arménie, **AZ** Azerbaïdjan, **BY** Bélarus, **KG** Kirghizistan, **KZ** Kazakhstan, **MD** République de Moldova, **RU** Fédération de Russie, **TJ** Tadjikistan, **TM** Turkménistan et tout autre État qui est un État contractant de la Convention sur le brevet eurasien et du PCT
- [X] **EP** **Brevet européen : AT** Autriche, **BE** Belgique, **CH et LI** Suisse et Liechtenstein, **CY** Chypre, **DE** Allemagne, **DK** Danemark, **ES** Espagne, **FI** Finlande, **FR** France, **GB** Royaume-Uni, **GR** Grèce, **IE** Irlande, **IT** Italie, **LU** Luxembourg, **MC** Monaco, **NL** Pays-Bas, **PT** Portugal, **SE** Suède et tout autre État qui est un État contractant de la Convention sur le brevet européen et du PCT TR TURQUIE
- [ ] **OA** **Brevet OAPI : BF** Burkina Faso, **BJ** Bénin, **CF** République centrafricaine, **CG** Congo, **CI** Côte d'Ivoire, **CM** Cameroun, **GA** Gabon, **GN** Guinée, **GW** Guinée-Bissau, **ML** Mali, **MR** Mauritanie, **NE** Niger, **SN** Sénégal, **TD** Tchad, **TG** Togo et tout autre État qui est un État membre de l'OAPI et un État contractant du PCT *(si une autre forme de protection ou de traitement est souhaitée, le préciser sur la ligne pointillée)* ....

Brevet national *(si une autre forme de protection ou de traitement est souhaitée, le préciser sur la ligne pointillée)* :

- [ ] **AE** Émirats arabes unis
- [ ] **AG** Antigua-et-Barbuda
- [ ] **AL** Albanie
- [ ] **AM** Arménie
- [ ] **AT** Autriche
- [X] **AU** Australie
- [ ] **AZ** Azerbaïdjan
- [ ] **BA** Bosnie-Herzégovine
- [ ] **BB** Barbade
- [ ] **BG** Bulgarie
- [ ] **BR** Brésil
- [ ] **BY** Bélarus
- [ ] **BZ** Belize
- [ ] **CA** Canada
- [ ] **CH et LI** Suisse et Liechtenstein
- [X] **CN** Chine
- [ ] **CR** Costa Rica
- [ ] **CU** Cuba
- [ ] **CZ** République tchèque
- [ ] **DE** Allemagne
- [ ] **DK** Danemark
- [ ] **DM** Dominique
- [ ] **DZ** Algérie
- [ ] **EE** Estonie
- [ ] **ES** Espagne
- [ ] **FI** Finlande
- [ ] **GB** Royaume-Uni
- [ ] **GD** Grenade
- [ ] **GE** Géorgie
- [ ] **GH** Ghana
- [ ] **GM** Gambie
- [ ] **HR** Croatie
- [ ] **HU** Hongrie
- [ ] **ID** Indonésie
- [ ] **IL** Israël
- [ ] **IN** Inde
- [ ] **IS** Islande
- [X] **JP** Japon
- [ ] **KE** Kenya
- [ ] **KG** Kirghizistan
- [ ] **KP** République populaire démocratique de Corée
- [ ] **KR** République de Corée
- [ ] **KZ** Kazakhstan
- [ ] **LC** Sainte-Lucie
- [ ] **LK** Sri Lanka
- [ ] **LR** Liberia
- [ ] **LS** Lesotho
- [ ] **LT** Lituanie
- [ ] **LU** Luxembourg
- [ ] **LV** Lettonie
- [ ] **MA** Maroc
- [ ] **MD** République de Moldova
- [ ] **MG** Madagascar
- [ ] **MK** Ex-République yougoslave de Macédoine
- [ ] **MN** Mongolie
- [ ] **MW** Malawi
- [ ] **MX** Mexique
- [ ] **MZ** Mozambique
- [ ] **NO** Norvège
- [ ] **NZ** Nouvelle-Zélande
- [ ] **PL** Pologne
- [ ] **PT** Portugal
- [ ] **RO** Roumanie
- [ ] **RU** Fédération de Russie
- [ ] **SD** Soudan
- [ ] **SE** Suède
- [ ] **SG** Singapour
- [ ] **SI** Slovénie
- [ ] **SK** Slovaquie
- [ ] **SL** Sierra Leone
- [ ] **TJ** Tadjikistan
- [ ] **TM** Turkménistan
- [ ] **TR** Turquie
- [ ] **TT** Trinité-et-Tobago
- [ ] **TZ** République-Unie de Tanzanie
- [ ] **UA** Ukraine
- [ ] **UG** Ouganda
- [X] **US** États-Unis d'Amérique
- [ ] **UZ** Ouzbékistan
- [ ] **VN** Viet Nam
- [ ] **YU** Yougoslavie
- [ ] **ZA** Afrique du Sud
- [ ] **ZW** Zimbabwe

Case réservée pour la désignation d'États qui sont devenus parties au PCT après la publication de la présente feuille :

- [ ] ....

Déclaration concernant les désignations de précaution : outre les désignations faites ci-dessus, le déposant fait aussi conformément à la règle 4.9.b) toutes les désignations qui seraient autorisées en vertu du PCT, à l'exception de toute désignation indiquée dans le cadre supplémentaire comme étant exclue de la portée de cette déclaration. Le déposant déclare que ces désignations additionnelles sont faites sous réserve de confirmation et que toute désignation qui n'est pas confirmée avant l'expiration d'un délai de 15 mois à compter de la date de priorité doit être considérée comme retirée par le déposant à l'expiration de ce délai. *(La confirmation (y compris les taxes) doit parvenir à l'office récepteur dans le délai de 15 mois.)*

REMPLI PAR RO

This Page Blank (uspto)

Cadre n° VI REVENDICATION [redacted] PRIORITÉ

☐ [redacted] autres revendications de priorité sont [redacted] quées dans le cadre supplémentaire.

Date de dépôt de la demande antérieure *(jour/mois/année)*	[redacted] méro de la demande antérieure	Lorsque la dema [redacted] térieure est une : demande nationale : pays	demande régionale :* office régional	demande internationale : office récepteur
(1) 17 mai 2000 (17.05.2000)	00 06882	FRANCE		
(2)				
(3)				

☒ L'office récepteur est prié de préparer et de transmettre au Bureau international une copie certifiée conforme de la ou des demandes antérieures *(seulement si la demande antérieure a été déposée auprès de l'office qui, aux fins de la présente demande internationale, est l'office récepteur)* indiquées ci-dessus au(x) point(s) : 1

** Si la demande antérieure est une demande ARIPO, il est obligatoire d'indiquer dans le cadre supplémentaire au moins un pays partie à la Convention de Paris pour la protection de la propriété industrielle pour lequel cette demande antérieure a été déposée (règle 4.10.b)ii)). Voir le cadre supplémentaire.*

Cadre n° VII ADMINISTRATION CHARGÉE DE LA RECHERCHE INTERNATIONALE

Choix de l'administration chargée de la recherche internationale (ISA) *(si plusieurs administrations chargées de la recherche internationale sont compétentes pour procéder à la recherche internationale, indiquer l'administration choisie; le code à deux lettres peut être utilisé) :*

ISA /

Demande d'utilisation des résultats d'une recherche antérieure; mention de cette recherche *(si une recherche antérieure a été effectuée par l'administration chargée de la recherche internationale ou demandée à cette dernière) :*

Date *(jour/mois/année)*	Numéro	Pays *(ou office régional)*
17.05.00	00 06882 FA	FR FA 592408

Cadre n° VIII BORDEREAU; LANGUE DE DÉPÔT

La présente demande internationale contient le **nombre de feuilles** suivant :

requête	:	04
description (sauf partie réservée au listage des séquences)	:	21
revendications	:	~~03~~ 4
abrégé	:	01
dessins	:	04
partie de la description réservée au listage des séquences	:	
Nombre total de feuilles	:	~~33~~ 34

Le ou les éléments cochés ci-après **sont joints** à la présente demande internationale :

1. ☒ feuille de calcul des taxes
2. ☐ pouvoir distinct signé
3. ☒ copie du pouvoir général; numéro de référence, le cas échéant : GPA01/00075
4. ☐ explication de l'absence d'une signature
5. ☒ document(s) de priorité indiqué(s) dans le cadre n° VI au(x) point(s) : 1
6. ☐ traduction de la demande internationale en (langue) :
7. ☐ indications séparées concernant des micro-organismes ou autre matériel biologique déposés
8. ☐ listage des séquences de nucléotides ou d'acides aminés sous forme déchiffrable par ordinateur
9. ☒ autres éléments *(préciser)* : Rapport de Recherche

Figure des dessins qui doit accompagner l'abrégé : 3

Langue de dépôt de la demande internationale : FRANCAIS

REMPLI PAR RO

Cadre n° IX SIGNATURE DU DÉPOSANT OU DU MANDATAIRE

À côté de chaque signature, indiquer le nom du signataire et, si cela n'apparaît pas clairement à la lecture de la requête, à quel titre l'intéressé signe.

CORLU Bernard (mandataire)

Réservé à l'office récepteur

1. Date effective de réception des pièces supposées constituer la demande internationale : 17 MAI 2001 (17-05-01)
2. Dessins : ☐ reçus : ☐ non reçus :
3. Date effective de réception, rectifiée en raison de la réception ultérieure, mais dans les délais, de documents ou de dessins complétant ce qui est supposé constituer la demande internationale :
4. Date de réception, dans les délais, des corrections demandées selon l'article 11.2) du PCT :
5. Administration chargée de la recherche internationale (si plusieurs sont compétentes) : ISA /
6. ☐ Transmission de la copie de recherche différée jusqu'au paiement de la taxe de recherche.

Réservé au Bureau international

Date de réception de l'exemplaire original par le Bureau international :

This Page Blank (uspto)

TRAITÉ DE COOPÉRATION EN MATIÈRE DE BREVETS

Expéditeur: le BUREAU INTERNATIONAL

PCT

AVIS INFORMANT LE DÉPOSANT DE LA COMMUNICATION DE LA DEMANDE INTERNATIONALE AUX OFFICES DÉSIGNÉS

(règle 47.1.c), première phrase, du PCT)

Destinataire:
CORLU, Bernard
Bull S.A.
PC 62A24
68, route de Versailles
Boîte postale 45
F-78434 Louveciennes Cedex
FRANCE

REÇU LE
04 DEC. 2001
PROPRIETE INTELLECTUELLE

Date d'expédition (jour/mois/année) 22 novembre 2001 (22.11.01)	
Référence du dossier du déposant ou du mandataire PCT 3884/BC	**AVIS IMPORTANT**

Demande internationale n°	Date du dépôt international (jour/mois/année)	Date de priorité (jour/mois/année)
PCT/FR01/01506	17 mai 2001 (17.05.01)	17 mai 2000 (17.05.00)

Déposant
BULL CP8 etc

1. Il est notifié par la présente qu'à la date indiquée ci-dessus comme date d'expédition de cet avis, le Bureau international a **communiqué**, comme le prévoit l'article 20, la demande internationale aux offices désignés suivants:
   US

   Conformément à la règle 47.1.c), troisième phrase, ces offices acceptent le présent avis comme preuve déterminante du fait que la communication de la demande internationale a bien eu lieu à la date d'expédition indiquée plus haut, et le déposant n'est pas tenu de remettre de copie de la demande internationale à l'office ou aux offices désignés.

2. Les offices désignés suivants ont renoncé à l'exigence selon laquelle cette communication doit être effectuée à cette date:
   AU,CN,EP,JP

   La communication sera effectuée seulement sur demande de ces offices. De plus, le déposant n'est pas tenu de remettre de copie de la demande internationale aux offices en question (règle 49.1)a-bis)).

3. Le présent avis est accompagné d'une copie de la demande internationale publiée par le Bureau international le
   22 novembre 2001 (22.11.01) sous le numéro WO 01/88705

RAPPEL CONCERNANT LE CHAPITRE II (article 31.2)a) et règle 54.2)

Si le déposant souhaite reporter l'ouverture de la phase nationale jusqu'à 30 mois (ou plus pour ce qui concerne certains offices) à compter de la date de priorité, **la demande d'examen préliminaire international** doit être présentée à l'administration compétente chargée de l'examen préliminaire international avant l'expiration d'un délai de 19 mois à compter de la date de priorité.

Il appartient exclusivement au déposant de veiller au respect du délai de 19 mois.

Il est à noter que seul un déposant qui est ressortissant d'un État contractant du PCT lié par le chapitre II ou qui y a son domicile peut présenter une demande d'examen préliminaire international (actuellement, tous les États contractants du PCT sont liés par le chapitre II).

RAPPEL CONCERNANT L'OUVERTURE DE LA PHASE NATIONALE (article 22 ou 39.1))

Si le déposant souhaite que la demande internationale procède en **phase nationale**, il doit, dans le délai de 20 mois ou de 30 mois, ou plus pour ce qui concerne certains offices, accomplir les actes mentionnés dans ces dispositions auprès de chaque office désigné ou élu.

Pour d'autres informations importantes concernant les délais et les actes à accomplir pour l'ouverture de la phase nationale, voir l'annexe du formulaire PCT/IB/301 (Notification de la réception de l'exemplaire original) et le Guide du déposant du PCT, volume II.

Bureau international de l'OMPI 34, chemin des Colombettes 1211 Genève 20, Suisse	Fonctionnaire autorisé J. Zahra
n° de télécopieur (41-22) 740.14.35	n° de téléphone (41-22) 338.91.11

This Page Blank (uspto)

TRAITE DE COOPERATION EN MATIERE DE BREVETS

Expéditeur: le BUREAU INTERNATIONAL

PCT

NOTIFICATION DE LA RECEPTION DE L'EXEMPLAIRE ORIGINAL

(règle 24.2.a) du PCT)

Destinataire:

CORLU, Bernard
Bull S.A.
PC 62A24
68, route de Versailles
Boîte postale 45
F-78434 Louveciennes Cedex
FRANCE

Date d'expédition (jour/mois/année) 26 juin 2001 (26.06.01)	**NOTIFICATION IMPORTANTE**
Référence du dossier du déposant ou du mandataire PCT 3884/BC	**Demande internationale no** PCT/FR01/01506

Il est notifié au déposant que le Bureau international a reçu l'exemplaire original de la demande internationale précisée ci-après.

Nom(s) du ou des déposants et de l'Etat ou des Etats pour lesquels ils sont déposants:

BULL CP8 (pour tous les Etats désignés sauf US)
FOUGEROUX, Nicolas etc. (pour US seulement)

Date du dépôt international	:	17 mai 2001 (17.05.01)
Date(s) de priorité revendiquée(s)	:	17 mai 2000 (17.05.00)
Date de réception de l'exemplaire original par le Bureau international	:	19 juin 2001 (19.06.01)
Liste des offices désignés	:	

EP :AT,BE,CH,CY,DE,DK,ES,FI,FR,GB,GR,IE,IT,LU,MC,NL,PT,SE,TR
National :AU,CN,JP,US

ATTENTION

Le déposant doit soigneusement vérifier les indications figurant dans la présente notification. En cas de divergence entre ces indications et celles que contient la demande internationale, il doit aviser immédiatement le Bureau international.

En outre, l'attention du déposant est appelée sur les renseignements donnés dans l'annexe en ce qui concerne

- [X] les délais dans lesquels doit être abordée la phase nationale
- [X] la confirmation des désignations faites par mesure de précaution
- [ ] les exigences relatives aux documents de priorité.

Une copie de la présente notification est envoyée à l'office récepteur et à l'administration chargée de la recherche internationale.

Bureau international de l'OMPI **34, chemin des Colombettes** **1211 Genève 20, Suisse** n° de télécopieur (41-22) 740.14.35	Fonctionnaire autorisé P. Blanchet (Fax 338.87.40) 17 n° de téléphone (41-22) 338.83.38

This Page Blank (uspto)

RENSEIGNEMENTS CONCERNANT LES DELAIS DANS LESQUELS DOIT ETRE ABORDEE LA PHASE NATIONALE

Il est rappelé au déposant qu'il doit aborder la "phase nationale" auprès de chacun des offices désignés indiqués sur la notification de la réception de l'exemplaire original (formulaire PCT/IB/301) en payant les taxes nationales et en remettant les traductions, telles qu'elles sont prescrites par les législations nationales.

Le délai d'accomplissement de ces actes de procédure est de **20 MOIS** à compter dela date de priorité ou, pour les Etats désignés qui ont été élus par le déposant dans une demande d'examen préliminaire international ou dans une élection ultérieure, de **30 MOIS** à compter de la date de priorité, à condition que cette électionait été effectuée avant l'expiration du 19e mois à compter de la date de priorité. Certains offices désignés (ou élus) ont fixé des délais qui expirent au-delà de 20 ou 30 mois à compter de la date de priorité. D'autres offices accordent une prolongation des délais ou un délai de grâce, dans certains cas moyennant le paiement d'une taxe supplémentaire.

En plus de ces actes de procédure, le déposant devra dans certains cas satisfaire à d'autres exigences particulières applicables dans certains offices. **Il appartient au déposant** de veiller à remplir en temps voulu les conditions requises pour l'ouverture de la phase nationale. La majorité des offices désignés n'envoient pas de rappel à l'approche de la date limite pour aborder la phase nationale.

Des informations détaillées concernant les actes de procédure à accomplir pour aborder la phase nationale auprès de chaque office désigné, les délais applicables et la possibilité d'obtenir une prolongation des délais ou un délai de grâce et toutes autres conditions applicables figurent dans le volume II du Guide du déposant du PCT. Les exigences concernant le dépôt d'une demande d'examen préliminaire international sont exposées dans le chapitre IX du volume I du Guide du déposant du PCT.

GR et ES sont devenues liées par le chapitre II du PCT le 7 septembre 1996 et le 6 septembre1997, respectivement, et peuvent donc être élues dans une demande d'examen préliminaire international ou dans une élection ultérieure présentée le 7 septembre 1996 (ou à une date postérieure) ou le 6 septembre 1997 (ou à une date postérieure), respectivement, quelle que soit la date de dépôt de la demande internationale (voir le second paragraphe, ci-dessus).

Veuillez noter que seul un déposant qui est ressortissant d'un Etat contractant du PCT lié par le chapitre II ou qui y a son domicile peut présenter une demande d'examen préliminaire international.

CONFIRMATION DES DESIGNATIONS FAITES PAR MESURE DE PRECAUTION

Seules les désignations expresses faites dans la requête conformément à la règle 4.9.a) figurent dans la présente notification. Il est important de vérifier si ces désignations ont été faites correctement. Des erreurs dans les désignations peuvent être corrigées lorsque des désignations ont été faites par mesure de précaution en vertu de la règle 4.9.b). Toute désignation ainsi faite peut être confirmée conformément aux dispositions de la règle 4.9.c) avant l'expiration d'un délai de 15 mois à compter de la date de priorité. En l'absence de confirmation, une désignation faite par mesure de précaution sera considérée comme retirée par le déposant. Il ne sera adressé aucun rappel ni invitation. Pour confirmer une désignation , il faut déposer une déclaration précisant l'Etat désigné concerné (avec l'indication de la forme de protection ou de traitement souhaitée) et payer les taxes de désignation et de confirmation. La confirmation doit parvenir à l'office récepteur dans le délai de 15 mois.

EXIGENCES RELATIVES AUX DOCUMENTS DE PRIORITE

Pour les déposants qui n'ont pas encore satisfait aux exigences relatives aux documents de priorité, il est rappelé ce qui suit.

Lorsque la priorité d'une demande nationale, régionale ou internationale antérieure est revendiquée, le déposant doit présenter une copie de cette demande antérieure, certifiée conforme par l'administration auprès de laquelle elle a été déposée ("document de priorité"), à l'office récepteur (qui la transmettra au Bureau international) ou directement au Bureau international, avant l'expiration d'un délai de 16 mois à compter de la date de priorité, étant entendu que tout document de priorité peut être présenté au Bureau international avant la date de publication de la demande internationale, auquel cas ce document sera réputé avoir été reçu par le Bureau international le dernier jour du délai de 16 mois (règle 17.1.a)).

Lorsque le document de priorité est délivré par l'office récepteur, le déposant peut, au lieu de présenter ce document, demander à l'office récepteur de le préparer et de le transmettre au Bureau international. La requête à cet effet doit être formulée avant l'expiration du délai de 16 mois et peut être soumise au paiement d'une taxe (règle 17.1.b)).

Si le document de priorité en question n'est pas fourni au Bureau international, ou si la demande adressée à l'office récepteur de préparer et de transmettre le document de priorité n'a pas été faite (et la taxe correspondante acquittée, le cas échéant) avant l'expiration du délai applicable mentionné aux paragraphes précédents, tout Etat désigné peut ne pas tenir compte de la revendication de priorité; toutefois, aucun office désigné ne peut décider de ne pas tenir compte de la revendication de priorité avant d'avoir donné au déposant la possibilité de remettre le document de priorité dans un délai raisonnable en l'espèce.

Lorsque plusieurs priorités sont revendiquées, la date de priorité à prendre en considération aux fins du calcul du délai de 16 mois est la date du dépôt de la demande la plus ancienne dont la priorité est revendiquée.

This Page Blank (uspto)

INPI
INSTITUT NATIONAL DE LA PROPRIETE INDUSTRIELLE

26 bis, rue de Saint Pétersbourg
75800 Paris Cedex 08
Téléphone : 01 53 04 53 04 Télécopie : 01 42 94 86 54

BREVET D'INVENTION
CERTIFICAT D'UTILITÉ

cerfa
N° 11354*01

Code de la propriété intellectuelle - Livre VI

REQUÊTE EN DÉLIVRANCE 1/2

Cet imprimé est à remplir lisiblement à l'encre noire   DB 540 W /260899

Réservé à l'INPI	
REMISE DES PIÈCES DATE 17 MAI 2000 LIEU 75 INPI PARIS F N° D'ENREGISTREMENT NATIONAL ATTRIBUÉ PAR L'INPI 0006882 DATE DE DÉPÔT ATTRIBUÉE PAR L'INPI	**1** NOM ET ADRESSE DU DEMANDEUR OU DU MANDATAIRE À QUI LA CORRESPONDANCE DOIT ÊTRE ADRESSÉE BULL S.A. CORLU Bernard - PC/58D20 68, route de Versailles 78434 LOUVECIENNES Cedex
Vos références pour ce dossier *(facultatif)* FR 3884/BC	

Confirmation d'un dépôt par télécopie ☒ N° attribué par l'INPI à la télécopie 2236 du 17 MAI 2000

2 NATURE DE LA DEMANDE	**Cochez l'une des 4 cases suivantes**		
Demande de brevet	☒		
Demande de certificat d'utilité	☐		
Demande divisionnaire	☐		
Demande de brevet initiale	N°	Date	/ /
ou demande de certificat d'utilité initiale	N°	Date	/ /
Transformation d'une demande de brevet européen *Demande de brevet initiale*	☐ N°	Date	/ /

3 TITRE DE L'INVENTION (200 caractères ou espaces maximum)

Procédé de sécurisation d'un langage du type à données typées, notamment dans un système embarqué et système embarqué de mise en œuvre du procédé.

4 DÉCLARATION DE PRIORITÉ OU REQUÊTE DU BÉNÉFICE DE LA DATE DE DÉPÔT D'UNE DEMANDE ANTÉRIEURE FRANÇAISE	Pays ou organisation Date / /	N°
	Pays ou organisation Date / /	N°
	Pays ou organisation Date / /	N°
	☐ **S'il y a d'autres priorités, cochez la case et utilisez l'imprimé «Suite»**	

5 DEMANDEUR		☐ **S'il y a d'autres demandeurs, cochez la case et utilisez l'imprimé «Suite»**
Nom ou dénomination sociale		BULL CP8
Prénoms		
Forme juridique		Société Anonyme
N° SIREN		3 2 9 5 5 6 1 4 6
Code APE-NAF		B 3 2 1
Adresse	Rue	BP 45 - 68, route de Versailles
	Code postal et ville	~~78430 LOUVECIENNES~~
Pays		~~France~~
Nationalité		~~Française~~
N° de téléphone *(facultatif)*		~~01.39.66.61.76~~
N° de télécopie *(facultatif)*		~~01.39.66.61.73~~
Adresse électronique *(facultatif)*		~~BERNARD.CORLU@BULL.NET~~

This Page Blank (uspto)

(12) DEMANDE INTERNATIONALE PUBLIÉE EN VERTU DU TRAITÉ DE COOPÉRATION EN MATIÈRE DE BREVETS (PCT)

(19) Organisation Mondiale de la Propriété Intellectuelle
Bureau international

(43) Date de la publication internationale
22 novembre 2001 (22.11.2001)

PCT

(10) Numéro de publication internationale
WO 01/88705 A1

(51) Classification internationale des brevets[7] : G06F 9/455, 9/445

(21) Numéro de la demande internationale : PCT/FR01/01506

(22) Date de dépôt international : 17 mai 2001 (17.05.2001)

(25) Langue de dépôt : français

(26) Langue de publication : français

(30) Données relatives à la priorité :
00/06882 17 mai 2000 (17.05.2000) FR

(71) Déposant *(pour tous les États désignés sauf US)* : **BULL CP8** [FR/FR]; 68, route de Versailles, Boîte postale 45, F-78430 Louveciennes (FR).

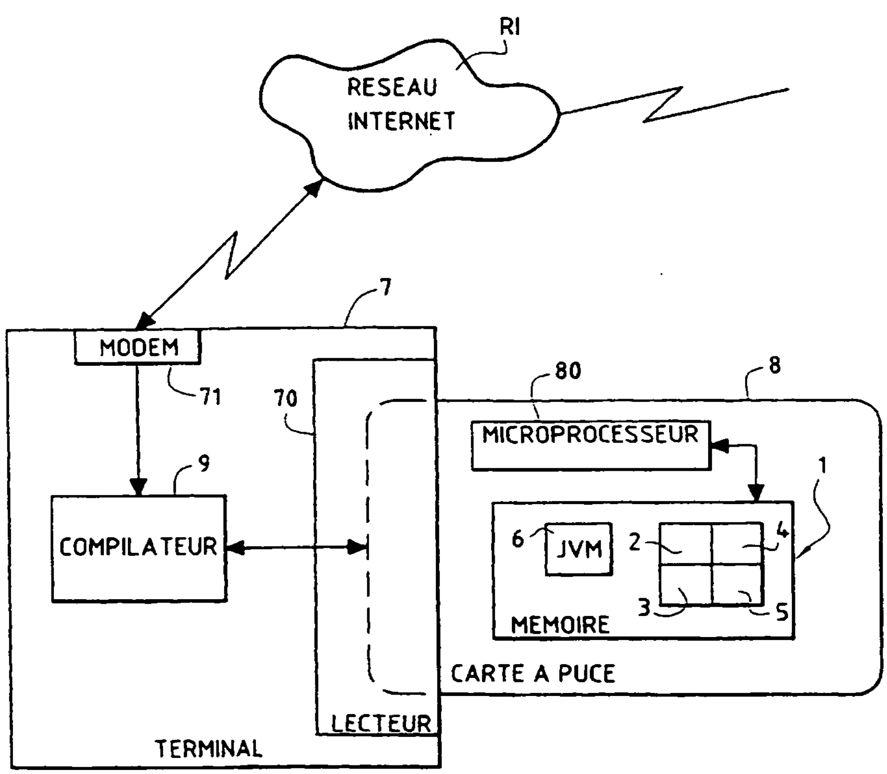
(72) **Inventeurs; et**

(75) **Inventeurs/Déposants** *(pour US seulement)* : **FOUGEROUX, Nicolas** [FR/FR]; 6, square Bainville, F-78150 Le Chesnay (FR). **LANDIER, Olivier** [FR/FR]; 138, avenue Félix Faure, F-75015 Paris (FR). **HAMEAU, Patrice** [FR/FR]; 18, rue de Belle Feuille, F-92100 Boulogne Billancourt (FR).

[Suite sur la page suivante]

(54) **Title:** METHOD FOR MAKING SECURE A TYPED DATA LANGUAGE IN PARTICULAR IN AN INTEGRATED SYSTEM AND INTEGRATED SYSTEM THEREFOR

(54) **Titre :** PROCEDE DE SECURISATION D'UN LANGAGE DU TYPE A DONNEES TYPEES, NOTAMMENT DANS UN SYSTEME EMBARQUE ET SYSTEME EMBARQUE DE MISE EN OEUVRE DU PROCEDE



1... STORAGE
8... SMART CARD
9... COMPILER
70... READER
80... MICROPROCESSOR
RI... INTERNET

(57) Abstract: The invention concerns a method and a microchip (8) integrated system for executing securely a sequence of instructions of a computer application in the form of objects or typed data, in particular written in JAVA language. The storage unit (1) is arranged into a first series of elementary stacks (2, 3) for recording the instructions. The method consists in associating with each data or typed object one or several so-called typing bits specifying the type. Said bits are recorded in a second series of stacks (4, 5) in one-to-one relationship with the stacks (2, 3) of the first series. Before executing the instructions of predetermined types, a continuous verification is carried out, prior to execution of said instructions, of the conformity between a type mentioned

WO 01/88705 A1

[Suite sur la page suivante]

(74) Mandataire : CORLU, Bernard; Bull S.A., PC 62A24, 68, route de Versailles, Boîte postale 45, F-78434 Louveciennes Cedex (FR).

(81) États désignés *(national)* : AU, CN, JP, US.

(84) États désignés *(régional)* : brevet européen (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR).

Publiée :
— *avec rapport de recherche internationale*

En ce qui concerne les codes à deux lettres et autres abréviations, se référer aux "Notes explicatives relatives aux codes et abréviations" figurant au début de chaque numéro ordinaire de la Gazette du PCT.

in the instructions and an expected type, indicated by the typing bits. In case of non-conformity, the execution is stopped.

(57) Abrégé : L'invention concerne un procédé et un système embarqué à puce électronique (8) pour l'exécution sécurisée d'une séquence d'instructions d'une application informatique se présentant sous la forme d'objects ou de données typées, notamment écrite en langage "JAVA". La mémoire (1) est organisée en une première série de piles élémentaires (2, 3) pour l'enregistrement des instructions. On associe à chaque donnée ou objet typé un ou plusieurs bits dits de typage spécifiant le type. Ces bits sont enregistrés dans une deuxième série de piles élémentaires (4, 5), en relation biunivoque avec les piles (2, 3) de la première série. Avant l'exécution d'instructions de types prédéterminés, il est procédé à une vérification en continu, préalable à l'exécution de ces instructions, de la concordance entre un type indiqué par celles-ci et un type attendu, indiqué par les bits de typage. En cas de non-concordance l'exécution et stoppée.

PROCEDE DE SECURISATION D'UN LANGAGE DU TYPE A DONNEES TYPEES, NOTAMMENT DANS UN SYSTEME EMBARQUE ET SYSTEME EMBARQUE DE MISE EN ŒUVRE DU PROCEDE

L'invention concerne un procédé de sécurisation dynamique d'un langage du type à données typées, notamment pour un système embarqué à puce électronique.

L'invention concerne encore un système embarqué à puce électronique pour la mise en œuvre du procédé.

Dans le cadre de l'invention, le terme "système embarqué" doit être compris dans son sens le plus général. Il concerne notamment toutes sortes de terminaux légers munis d'une puce électronique, et plus particulièrement les cartes à puce proprement dites. La puce électronique est munie de moyens d'enregistrement et de traitement de données numériques, par exemple un microprocesseur pour ces derniers moyens.

Pour fixer les idées, et sans que cela limite en quoi que ce soit sa portée, on se placera ci-après dans le cas de l'application préférée de l'invention, à savoir les applications à base de cartes à puce, sauf mention contraire.

De même, bien que divers langages informatiques, tels les langages "ADA" ou "KAMEL" (tous deux étant des marques déposées), sont du type dit à données ou objets typés, un des langages les plus utilisés dans le domaine préféré de l'invention étant le langage de type objet "JAVA" (marque déposée), ce langage sera pris comme exemple ci-après, pour décrire en détail le procédé de l'invention.

Enfin, le terme "sécurisation" doit lui aussi être compris dans un sens général. Notamment, il concerne aussi bien ce qui a trait au concept de confidentialité des données manipulées qu'au concept d'intégrité, matérielle et/ou logicielle, des composants présents dans le système embarqué.

Avant de décrire plus avant l'invention, il est tout d'abord utile de rappeler brièvement les principales caractéristiques du langage "JAVA", notamment dans un environnement du type carte à puce.

Ce dernier langage présente en particulier l'avantage d'être multiplateformes : il suffit que la machine dans laquelle s'exécute l'application écrite en langage "JAVA" soit munie d'un minimum de ressources informatiques spécifiques, notamment d'une pièce de logiciel appelé "machine virtuelle JAVA" pour l'interprétation d'une suite de séquences d' "opcodes" d'instructions de 8 bits, appelées "bytecode" ou "p-code" (pour "program code" ou code programme). Le "p-code" est enregistré dans des positions de mémoire des moyens d'enregistrement de données précités. Plus précisément, dans le cas du langage "JAVA", la zone occupée par les positions de mémoire, d'un point de vue logique, se présente sous une configuration connue sous le nom de pile.

Dans le cas d'une carte à puce, celle-ci intègre la "machine virtuelle JAVA" (enregistrée dans ses moyens de mémoire) et fonctionne en interprétant un langage basé sur la séquence d'opcodes précitée. Le code exécutable ou "p-code" résulte d'une compilation préalable. Le compilateur est agencé pour que le langage transformé obéisse à un format déterminé et respecte un certain nombre de règles fixées *a priori*.

Les "opcodes" peuvent recevoir des valeurs d'éléments les suivants dans une séquence du "p-code", ces éléments sont alors appelés paramètres. Les opcodes peuvent aussi recevoir des valeurs en provenance de la pile. Ces éléments constituent alors des opérandes.

Selon une autre caractéristique du langage "JAVA", il est mis en œuvre des éléments connus sous les noms de "classes" et de "méthodes". Lors de l'exécution d'une méthode donnée, la machine virtuelle retrouve le "p-code" correspondant. Ce "p-code" identifie des opérations spécifiques à effectuer par la machine virtuelle. Une pile particulière est nécessaire pour le traitement de variables dites locales, d'opérations arithmétiques ou pour l'invocation d'autres méthodes.

Cette pile sert de zone de travail pour la machine virtuelle. Pour optimiser les performances de la machine virtuelle, la largeur de la pile est généralement fixée pour un type primitif donné.

Dans cette pile deux grands types d'objets peuvent être manipulés :

- des objets de type dit "primitif", ceux connus sous les dénominations "*int*" (pour entier long : 4 octets), "*short*" (pour entier court : 2 octets), "*byte*" (octet), "*boolean*" (objet booléen) ; et
- des objets de type dit "référence" (tableaux d'objets de type primitif, instances de classes).

La différence fondamentale entre ces deux types d'objets est que seule la machine virtuelle attribue une valeur à des objets de type référence et les manipule.

Les objets références peuvent être vus comme des pointeurs vers des zones mémoires de la carte à puce (références physiques ou logiques).

Le langage "JAVA", dont les principales caractéristiques viennent d'être succinctement rappelées, se prête particulièrement bien aux applications mettant en jeu des interconnexions avec le réseau Internet et son grand succès est d'ailleurs lié au fort développement des applications Internet.

D'un point de vue sécurité, il présente aussi un certain nombre d'avantages. Tout d'abord, le code exécutable ou "p-code" résulte d'une compilation préalable. Le compilateur peut donc être agencé, comme il a été indiqué, pour que le langage transformé obéisse à un format déterminé et respecte un certain nombre de règles fixées *a priori*.

Une de ces règles est qu'une application donnée soit confinée à l'intérieur de ce qui est appelé une "sand box" (ou "boite noire"). Les instructions et/ou données associées à une application déterminée sont mémorisées dans des positions de mémoire des moyens d'enregistrement de données. Dans le cas du langage "JAVA", d'un point de vue logique, la configuration de ces moyens d'enregistrement de données prend la forme d'une pile. Le confinement dans une "sand box" signifie en pratique que les

instructions précitées ne peuvent pas adresser des positions mémoires en dehors de celles affectées à ladite application, sans y être autorisées expressément.

Cependant, une fois chargé en mémoire, des problèmes de sécurité peuvent se poser si le "p-code" a été altéré ou si son format n'a pas respecté les spécifications de la machine virtuelle. Aussi, dans l'art connu, notamment lorsqu'il s'agit d'applications, par exemple des "applets" (appliquettes), téléchargées via le réseau Internet, le code compilé, c'est-à-dire le "p-code" est vérifié par la machine virtuelle. Cette dernière est habituellement associée à un navigateur de type "WEB" dont est muni le terminal connecté au réseau Internet. Pour ce faire, la machine virtuelle est elle-même associée à une pièce de logiciel particulière ou vérificateur.

Cette vérification peut s'effectuer en mode dit "off-line", c'est-à-dire hors connexion, ce qui ne pénalise pas le traitement de l'application, notamment d'un point de vue coût de communication.

On est ainsi sûr, après que la vérification soit effectuée, que le "p-code" n'est pas endommagé et est conforme au format et aux règles préétablis. On est aussi sûr, dans ces conditions, que lors de l'exécution du "p-code", il n'y aura pas de détérioration du terminal dans lequel il s'exécute.

Cependant, ce procédé n'est pas sans inconvénients, en particulier dans le cadre des applications visées préférentiellement par l'invention.

Tout d'abord, le vérificateur précité nécessite à lui seul une quantité de mémoire relativement importante, de l'ordre de plusieurs MO. Cette valeur élevée ne présente pas de problèmes particuliers si le vérificateur est enregistré dans un micro-ordinateur ou un terminal similaire disposant de ressources mémoires élevées. Cependant, lorsque l'on envisage d'utiliser un terminal de traitement de données possédant des ressources informatiques plus limitées, *a fortiori* une carte à puce, il n'est pas envisageable, d'un point de vue pratique, compte tenu des technologies actuellement disponibles, d'implémenter le vérificateur dans ce type de terminal.

On doit également noter que la vérification est d'un type que l'on peut qualifier de "statique", car effectuée une fois pour toute, avant l'exécution du "p-code". Lorsqu'il s'agit d'un terminal du type micro-ordinateur, notamment lorsque ce dernier est maintenu déconnecté lors de l'exécution du "p-code" vérifié au préalable, cette dernière caractéristique ne pose pas de problèmes particuliers. En effet, il n'existe pas de risques importants, d'un point de vue sécurité, car le terminal reste habituellement sous le contrôle de son opérateur.

Tel n'est pas le cas pour un système embarqué mobile, notamment pour une carte à puce. En effet, si le "p-code", même vérifié, est ensuite chargé dans les moyens d'enregistrement de données de la carte à puce, il peut subir *a posteriori* des altérations. En général, la carte à puce, ce par nature, n'est pas destinée à demeurer en permanence dans le terminal à partir duquel l'application a été chargée. A titre d'exemple non limitatif, la carte à puce peut être soumise à un rayonnement ionisant qui altère physiquement des positions de mémoire. Il est possible également d'altérer le "p-code" au moment de son téléchargement dans la carte à puce, à partir du terminal.

Il s'ensuit que, si le "p-code" est altéré, notamment dans un but malveillant, il est possible d'effectuer une opération dite de "dump" (duplication) de zones de mémoires et/ou de mettre en péril le bon fonctionnement de la carte à puce. Il devient ainsi possible, par exemple, et malgré la disposition dite de "sand box" précitée, d'avoir accès à des données confidentielles, ou pour le moins non autorisées, ou d'attaquer l'intégrité d'une ou plusieurs applications présentes sur la carte à puce. Enfin, si la carte à puce est connectée au monde extérieur, les dysfonctionnements provoqués peuvent se propager à l'extérieur de la carte à puce.

L'invention vise à pallier les inconvénients des procédés et dispositifs de l'art connu, et dont certains viennent d'être rappelés.

L'invention se fixe pour but un procédé de sécurisation dynamique d'applications en langage du type à données typées dans un système embarqué.

Elle se fixe également pour but un système pour la mise en œuvre de ce procédé.

Pour ce faire, selon une première caractéristique, un élément d'information binaire comprenant un ou plusieurs bits, que l'on appellera ci-après "élément d'information de type", est associé à chaque objet manipulé par la machine virtuelle, dans le cas du langage "JAVA" précité. De façon plus générale, un élément d'information de type est associé à chaque donnée typée manipulée dans un langage donné, du type à objets ou données typés.

Selon une autre caractéristique, les éléments d'information de type sont stockés physiquement dans des zones de mémoire particulières des moyens de mémorisation du système embarqué à puce électronique.

Selon une autre caractéristique encore la machine virtuelle, toujours dans le cas du langage "JAVA" vérifie lesdits éléments d'information de type lors de certaines opérations d'exécution du "p-code", telles la manipulation d'objet dans la pile, etc., opérations qui seront précisées ci-après. De façon plus générale également, pour un autre langage, le processus est similaire et il est procédé à une étape de vérification des éléments d'information de type. On constate donc que, de façon avantageuse, ladite vérification est d'un type que l'on peut appeler dynamique, puisqu'effectuée en temps réel lors de l'interprétation ou de l'exécution du code.

La machine virtuelle, ou ce qui en tient lieu pour un langage autre que le langage "JAVA", vérifie, en continu et avant ladite exécution d'une instruction ou d'une opération, que l'élément d'information de type correspond bien au type attendu de l'objet ou de la donnée typé à manipuler. Lorsqu'un type incorrect est détecté, des mesures sécuritaires sont prises afin de protéger la machine virtuelle et/ou d'empêcher toutes opérations non

conformes et/ou dangereuses pour l'intégrité du système embarqué à puce électronique.

Selon une première variante de réalisation supplémentaire du procédé selon l'invention, lesdits éléments d'information de type sont également utilisés avantageusement pour permettent la gestion de piles de largeurs variables, ce qui permet d'optimiser l'espace mémoire du système embarqué à puce électronique, dont les ressources de ce type sont, par nature, limitées, comme il a été rappelé.

Selon une deuxième variante de réalisation supplémentaire, cumulable avec la première, les éléments d'information de type sont également utilisés, en y adjoignant un ou plusieurs bit(s) d'information supplémentaire(s), utilisés comme "drapeau" ("flags" selon la terminologie anglo-saxonne), pour marquer les objets ou les données typées. Ce marquage est alors utilisé pour indiquer si ces derniers éléments sont utilisés ou non, et dans ce dernier cas, s'ils peuvent être effacés de la mémoire, ce qui permet également de gagner de la place mémoire.

L'invention a donc pour objet principal un procédé pour l'exécution sécurisée d'une séquence d'instructions d'une application informatique se présentant sous la forme de données typées enregistrées dans une première série d'emplacements déterminés d'une mémoire d'un système informatique, notamment un système embarqué à puce électronique, caractérisé en ce que des données supplémentaires dites éléments d'information de type sont associés à chacune desdites données typées, de manière à spécifier le type de ces données, en ce que lesdits éléments d'information de type sont enregistrés dans une deuxième série d'emplacements de mémoire déterminés de ladite mémoire de système informatique, et en ce que, avant l'exécution d'instructions d'un type prédéterminé, il est procédé à une vérification en continu, préalable à l'exécution d'instructions prédéterminées, de la concordance entre un type indiqué par ces instructions et un type attendu indiqué par lesdits éléments d'information de type enregistrés dans ladite deuxième série d'emplacement

de mémoire, de manière n'autoriser ladite exécution qu'en cas de concordance entre lesdits types.

L'invention a encore pour objet un système embarqué à puce électronique pour la mise en œuvre de ce procédé.

L'invention va maintenant être décrite de façon plus détaillée en se référant aux dessins annexés, parmi lesquels :

- les figures 1A à 1G illustrent les principales étapes d'une exécution correcte d'un exemple de "p-code" dans une mémoire à pile associée à des zones de mémoire spécifiques stockant des données dites éléments d'information de type selon l'invention ;
- les figures 2A et 2B illustrent schématiquement des étapes d'exécution de ce même code, mais contenant une altération menant à une exécution incorrecte et une détection de cette altération par le procédé de l'invention ; et
- la figure 3 illustre schématiquement un système comprenant une carte à puce pour la mise en œuvre du procédé selon l'invention.

Dans ce qui suit, sans en limiter en quoi que ce soit la portée, on se placera ci-après dans le cadre de l'application préférée de l'invention, sauf mention contraire, c'est-à-dire dans le cas d'un système embarqué à puce électronique intégrant une machine virtuelle "JAVA" pour l'interprétation de "p-code".

Comme il a été rappelé dans le préambule de la présente description, lors de l'exécution d'une méthode donnée, la machine virtuelle retrouve le "p-code" correspondant. Ce "p-code" identifie des opérations spécifiques à effectuer par la machine virtuelle. Une pile particulière est nécessaire pour le traitement de variables dites locales, d'opérations arithmétiques ou pour l'invocation d'autres méthodes.

La pile sert de zone de travail pour la machine virtuelle. Pour optimiser les performances de la machine virtuelle, la largeur de la pile est généralement fixée pour un type primitif donné.

Comme il a été également rappelé, dans cette pile deux grands types d'objets peuvent être manipulés :

- des objets de type dit "primitif", ceux connus sous les dénominations "*int*" (pour entier long : 4 octets), "*short*" (pour entier court : 2 octets), "*byte*" (octet), "*boolean*" (objet booléen) ; et
- des objets de type dit "référence" (tableaux d'objets de type primitif, instances de classes).

C'est ce dernier type d'objets qui pose le plus de problème, d'un point de vue sécurité, puisqu'il existe des possibilités, comme indiqué précédemment, de les manipuler de façon artificielle et de créer ainsi des dysfonctionnements de natures diverses.

Ils existent plusieurs types d' "opcodes", et notamment :

- la création d'un objet de type primitif (par exemple les opcodes dénommés "*bipush*" ou "*iconst*") ;
- l'exécution d'opérations arithmétiques sur des objets de type primitif (par exemple les "opcodes" dénommés "*iadd*" ou "*sadd*") ;
- la création d'un objet référence (par exemple les "opcodes" dénommés "*new*", "*newarray*" ou "*anewarray*").
- la gestion de variables locales (par exemple les "opcodes" dénommés "*aload*", "*iload*" ou "*istore*") ; et
- la gestion de variables de classes (par exemple les "opcodes" dénommés "*getstatic_a*" ou "*putfield_i*").

Chaque "opcode" qui utilise des objets placés en pile est typé afin de s'assurer que son exécution puisse être contrôlée. Généralement la(les) première(s) lettre(s) des "opcodes" indique(nt) le type utilisé. A titre d'exemple, et pour fixer les idées, (la ou les première(s) lettre(s) étant graissée pour mettre en évidence cette disposition), on peut citer les "opcodes" suivants :

- "***a****load*" pour les objets références ;
- "***i****load*" pour les entiers ; et
- "***ia****load*" pour les tableaux d'entiers.

Dans ce qui suit, par mesure de simplification la "machine virtuelle JAVA" sera appelée JVM.

Selon une première caractéristique du procédé selon l'invention, des éléments d'information de type sont stockés dans une zone mémoire sous la forme, chacun, d'un ou de plusieurs bits. Chacun de ces éléments d'information de type caractérise un objet manipulé par la JVM. On associe notamment un élément d'information de type à :

- chaque objet empilé dans la zone de donnée de la pile ;
- chaque variable locale (variable dont la portée ne dépasse pas le cadre d'une méthode) ; et
- à chaque objet de ce qui est appelé le "heap", c'est-à-dire une zone de mémoire stockant les objets dits "référence", chaque tableau et chaque variable globale.

Cette opération peut être appelée "typage" des objets. Selon une deuxième caractéristique du procédé de l'invention, la JVM vérifie le typage dans les cas suivants :

- lorsqu'un "opcode" manipule un objet stocké dans la pile ;
- récupère un objet dans la zone du "heap" ou dans celle des variables locales pour le placer en pile ;
- modifie un objet dans la zone du "heap" ou dans celle des variables locales ; et
- lors de l'invocation d'une nouvelle méthode, lorsque les opérandes sont comparés à la signature de la méthode.

Selon une autre caractéristique du procédé de l'invention, la JVM vérifie, avant l'exécution des opérations ci-dessus, que leurs types correspondent bien à celui attendu (c'est-à-dire ceux donnés par les "opcode" à effectuer).

Dans le cas de la détection d'un type incorrect, des mesures sécuritaires sont prises afin de protéger la JVM et/ou d'empêcher toutes opérations illégales ou dangereuses pour l'intégrité du système, tant d'un point de vue logique que matériel.

Pour mieux expliciter le procédé selon l'invention, on va maintenant le détailler en considérant un exemple particulier de code source en langage "JAVA".

On suppose également que la JVM est associée à une pile de 32 bits comportant au plus 32 niveaux et supportant les types primitifs (par exemple "*int*", "*short*", "*byte*", "*boolean*" et "*object reference*")

Le typage de la pile, selon l'une des caractéristiques de l'invention, peut alors être réalisé à l'aide d'éléments d'information de type de longueur 3 bits, conformément à la TABLE I placée en fin de la présente description. Les valeurs portées dans la TABLE I sont naturellement arbitraires. D'autres conventions pourraient être prises sans sortir du cadre de l'invention.

Le code source "JAVA" qui va être considéré ci-après à titre d'exemple particulier est le suivant :

Source "JAVA" (1) :

```
Public void method(){
int[] buffer;               //Déclaration
buffer=new int[2] ;         // création d'un tableau d'entiers de 2
éléments
buffer[1]=5 ;               // initialisation du tableau avec la valeur 5
}
```

Après un passage dans un compilateur approprié, un fichier "classe" contenant le "p-code" (2) correspondant au code source ci-dessus (1) est obtenu. Il se présente comme suit :

"p-code" (2) :

```
iconst_2   // Push int constant 2
newarray   T_INT
```

```
astore_1   int[] buffer
aload_1    int[] buffer
iconst_1   // Push int constant 1
iconst_5   // Push int constant 5
iastore
return
```

Comme il est bien connu de l'homme de métier, les trois premières lignes correspondent à la création du tableau précité (voir code source (1)). Les cinq dernières lignes correspondent à l'initialisation de ce tableau

On va maintenant illustrer en détail les étapes d'une exécution correcte du "p-code" ci-dessus. Puisque le "p-code" est un langage de type interprété, les lignes successives sont lues les unes après les autres et les étapes précitées correspondent à l'exécution de ces lignes, avec éventuellement l'exécution d'itérations et/ou de branchements. Dans ce qui suit, les différentes lignes de code sont graissées pour les mettre en évidence.

Exécution correcte :

Etape 1 : "***iconst_2***"

La figure 1A illustre de façon schématique l'étape d'exécution de ce "p-code". On a représenté, sous la référence 1, la mémoire du système embarqué à puce électronique (non représenté). De façon plus précise, cette mémoire 1 est divisée en quatre parties principales, deux étant communes à l'art connu : la zone dite "*zone data*" (données) 2*a* et la zone dite "*zone variable locale*" 3*a*. Ces zones, 2*a* et 3*a*, constituent la pile proprement dite de la machine virtuelle "JAVA" (JVM) que l'on appellera ci-après par simplification "*pile de la JVM*".

A ces zones sont associées des zones de mémoire, 4*a* et 5*a*, respectivement, spécifiques à l'invention, que l'on appellera zones de

"*Typage*". Selon un des aspects de l'invention, les zones de mémoire, 4*a* et 5*a*, sont destinées à stocker des éléments d'information de type (de longueur 3 bits dans l'exemple décrit) associés aux données stockées dans les zones 2*a* et 3*a*, respectivement, dans des emplacements de mémoire en relation biunivoque avec les emplacements de mémoire de ces zones. L'organisation logique de ces zones de mémoire est du type dit "pile" comme rappelé. Aussi, elles ont été représentées sous la forme de tableaux de dimensions *cxl*, avec *c* nombre de colonnes et *l* nombre de lignes, c'est-à-dire la "hauteur" de la pile ou niveau (qui peut varier à chaque étape de l'exécution d'un "p-code"). Dans l'exemple, *c*=4 pour les zones "*zone data*" 2*a* et "*zone variable locale*" 3*a* (chaque colonne correspondant à une position de mémoire de 4 octets, soit au total 32 bits), et *c*=3 pour les zones de "*typage*", 4*a* et 5*a*, (chaque colonne correspondant à une position de mémoire de 1 bit). Sur la figure 1A, le nombre de lignes représenté (ou numéro de niveau : 1 à 32 maximum dans l'exemple décrit) est égal à 2 pour toutes les zones de mémoire. Chacune des zones de mémoire, 2*a* à 5*a*, constitue donc une pile élémentaire.

On doit bien comprendre cependant que, physiquement, les positions de mémoires précitées peuvent être réalisées à base de divers circuits électroniques : cellules de mémoire vive, registres, etc. De même, elles ne sont pas forcément contiguës dans l'espace mémoire 1. La figure 1A ne constitue qu'une représentation schématique de l'organisation logique en piles de la mémoire 1.

L' "opcode" à exécuter pendant cette première étape n'a ni paramètre, ni opérande. La valeur entière 2 (soit "0002") est placée dans la pile : au niveau 1 (ligne inférieure dans l'exemple) de la zone 2*a*. La zone de "*Typage*" correspondante 4*a* est mise à jour.

D'après les conventions de la TABLE I, la valeur "*int*" (entier) "000" (en bits) est placée dans la zone de "*Typage*" 4*a*, également au niveau 1 (ligne inférieure). Aucune valeur n'est placée dans la "*zone variable locale*" 3*a*. Il en est de même de la zone de "*Typage*" correspondante 5*a*.

Etape 2 : **newarray** T_INT

L'étape correspondante est illustrée par la figure 1B.

Les éléments communs à la figure 1A portent les mêmes références numériques et ne seront re-décrits qu'en tant que de besoin. Seule la valeur littérale associée aux valeurs numériques est modifiée. Elle est identique à celle de la figure correspondante, soit *b* dans le cas de la figure 1B, de manière à caractériser les modifications successives des contenus des zones de mémoire. Il en sera de même pour les figures suivantes 1C à 1G.

L' "opcode" à exécuter pendant cette deuxième étape a pour paramètre le type de tableau à créer (soit type "*int*").

Cet "opcode" a pour opérande une valeur qui doit être de type "*int*", correspondant à la taille du tableau à créer (soit 2).

La vérification de la zone de "Typage" (à l'état 4*a*) indique un type correct. L'exécution est donc possible.

Un objet référence est créé dans la "Pile JVM" : par exemple la valeur (arbitraire) de quatre octets "1234" est placée dans les positions de mémoire de la "*zone variable locale*" (niveau 1). Puisqu'il s'agit d'un objet de type référence, la valeur "100" (en bits) est placée dans la zone de "Typage" correspondante 5*b* (niveau 1).

Aucune valeur n'est placée dans la zone de mémoire 3*b*, ni dans la zone de "*Typage*" 5*b*.

Etape 3 : **astore_1** **int[] buffer**

Cette étape est illustrée par la figure 1C.

L' "opcode" a pour opérande une valeur qui doit être de type "Objet référence". La vérification de la zone de "Typage" (à l'état 4*b*) indique un type correct. L'exécution est donc possible.

L'objet référence est déplacé vers la "*zone variable locale*" 3*c* : emplacement 1 (niveau 1).

Les zones de "Typage", 4*c* et 5*c* sont mise à jour : la valeur "100" (en bits) est déplacée du niveau 1 de la zone 4*c* vers le niveau 1 de la zone 5*c*.

Etape 4 : **aload_1** **int[] buffer**

Cette étape est illustrée par la figure 1D.

Cet "opcode" a pour objet d'empiler l'objet référence "1234", stocké dans la *"zone variable locale"* 3*d*, au niveau 1 de la *"zone data"* 2*d*, c'est-à-dire dans les positions de mémoire de la ligne inférieure de cette zone.

La vérification de la zone de "Typage" (à l'état 5*c*) indique un type correct. L'exécution est donc possible.

L'objet référence "1234" est placé dans la *"zone data"* 2*d*.

Les zones de "Typage" 4*d* et 5*d* sont mises à jour et stockent toutes deux, dans les emplacements de mémoire correspondants, la valeur "100" (en bits), représentative d'un type "Objet référence".

Etape 5 : **iconst_1** **// Push int constant 1**

Cette étape est illustrée par la figure 1E.

L' "opcode" à exécuter pendant cette étape n'a ni paramètre ni opérande. La valeur entière 1 (soit "0001") est placée dans la pile : emplacement 2 (niveau 2) de la *"zone data"* 2*e*. La zone de *"Typage"* correspondante 4*e* est mise à jour, également au niveau 2 (le niveau 1 reste inchangé : valeur "1000"). La valeur *"int"* (entier) "000" (en bits) est placée dans la zone de *"Typage"* 4^e (niveau 2). Les zones 3*e* et 5*e* restent inchangées.

Etape 6 : **iconst_5** **// Push int constant 5**

Cette étape est illustrée par la figure 1F.

L' "opcode" à exécuter pendant cette étape n'a ni paramètre ni opérande. La valeur entière 1 (soit "0001") est placée dans la pile : niveau 3 de la *"zone data"* 2*f*. La zone de *"Typage"* correspondante 4*f* est mise à jour, également au niveau 3 (les niveaux 1 et 2 restent inchangés : valeurs "1000" et "000" respectivement). La valeur *"int"* (entier) "000" (en bits) est placée dans la zone de *"Typage"* 4*f*. Les zones 3*f* et 5*f* restent inchangées.

Etape 7 : **iastore**

Cette étape est illustrée par la figure 1G.

Cet "opcode" a pour opérande une valeur de type "*int*", un index de type "*int*" et un objet référence de type tableau.

La vérification de la zone de "*Typage*" (à l'état 4*f* : niveau 3) indique un type correct. L'exécution est donc possible.

La valeur est stockée dans l'objet référence à l'index donné.

Etape 7 : **return**

Cet "opcode" indique la fin de la méthode, la pile doit alors être vide.

En considérant de nouveau le même "p-code" (voir (2), obtenu après compilation du code source (1)), on va maintenant détailler un exemple d'exécution incorrecte.

<u>Exécution incorrecte</u> :

A l'étape que l'on nommera 4' (correspondant à l'étape 4 : figure 1D). Il est supposé que le "p-code" a été altéré et que l' "opcode" :

"aload_1 int [] buffer" ,

a été remplacé, par exemple, par l' "opcode" suivant :

"iipush 0x5678",

instruction dans laquelle " 0x" indique une valeur hexadécimale.

Comme illustré par la figure 2A, cet "opcode", de type objet de référence, stocké au niveau 1 de la "*zone variable locale*" 3*a*', a pour objet d'empiler un entier de valeur "5678" dans la pile, dans la "*zone data*" 2'*a*.

La zone de "*Typage*" 4a' va être mise à jour. Il s'ensuit que les niveaux 1 des zones de "*Typage*", 4*a*' et 5*a*', vont tous deux contenir la valeur "100" (en bits), c'est-à-dire une valeur associée à un "Objet référence". Cette configuration particulière est illustrée par la figure 2A.

L'exécution se poursuit normalement comme dans le cas précédemment illustré par référence aux figures 1E et 1F.

Etape 5' : **iconst_1 // Push int constant 1**

Etape 6' : **iconst_5** **// Push int constant 5**

L'état des zones de la "*pile de la JVM*", "*zone variable locale*" 3b' et "*zone data*" 2b', est illustré par la figure 2B. de façon plus précise la "*zone data*" 2b' enregistre, au niveau 1, la valeur entière "5678", au niveau 2, la valeur entière "0001" et au niveau 3, la valeur entière "0005". La "*zone variable locale*" 3a' est restée inchangée. Il en est de même de la zone de "*Typage*" correspondante 5*a*'. Par contre, la zone de "*Typage*" 4*b*' est mise à jour et les valeurs suivantes sont enregistrées aux niveaux respectifs 1 à 3 : "100", "000" et "000" (en bits).

Etape 7' : **iastore**

Cet "opcode" a pour opérande une valeur de type "*int*", un index de type "*int*" et un objet référence de type tableau.

La vérification de la zone de "*Typage*" (niveau 1 de la zone, à l'état 4*b*') indique que le code détecté est incorrect. En effet, un entier ("*int*" ; code "000") est attendu à la place d'un "Objet référence" (code "100").

La JVM détecte donc la présence d'un "opcode" illégal menaçant la sécurité du système. L'exécution normale de la séquence d'instructions en cours est interrompue et remplacée par l'exécution d'instructions correspondant à des mesures sécuritaires pré-programmées : signal d'alerte, etc.

On a supposé jusqu'à présent que la largeur (ou taille) de la "*pile de la JVM*" ; que ce soit celle de la "*zone data*" ou la "*zone variable locale*", était fixe, ce qui est généralement le cas dans l'art connu. Dans l'exemple décrit, on a supposé que chaque emplacement de mémoire compte quatre octets (soit 32 bits). Cependant, une telle disposition s'avère pénalisante en terme de capacité de mémoire. En effet, d'une application logicielle à l'autre, voire à l'intérieur d'une même application, le nombre d'octets nécessaire pour chaque instruction est variable. Comme il a été indiqué, l'agencement les piles élémentaires des "*zone data*" et "*zone variable locale*" telles qu'illustrées par les figures 1A à 1G, ou 2A à 2B, ne représentent qu'une vue logique de l'espace mémoire 1. Il est donc tout à fait possible de conserver

une architecture logique du type pile, même si les emplacements de mémoire, successifs ou non, sont de longueurs variables, voire même si les différentes positions (cellules) de mémoire sont physiquement dispersées.

Aussi, selon une première variante supplémentaire du procédé selon l'invention, les éléments d'information de type permettent aussi de déterminer la largeur instantanée nécessaire, en positions de mémoire, des zones de la "*pile de la JVM*". Il suffit, pour ce faire, que les codes enregistrés dans les zones de "*Typage*" de la mémoire soient associés, en tout ou partie, à une information caractérisant la largeur de la pile précitée. A titre d'exemple non limitatif, il peut s'agir de bits supplémentaires, ajoutés aux codes de typage, ou d'une combinaison de bits non utilisée de ces codes. Dans le premier cas, si la largeur de la pile peut varier, toujours à titre d'exemple, entre 1 et 4 octets, il suffit de 2 bits supplémentaires pour caractériser les largeurs suivantes :

Configuration binaire	00	01	10	11
Largeur en octets	1	2	3	4

Cette disposition, qui permet d'optimiser l'espace mémoire en fonction des applications à exécuter, conduit à un gain de place de mémoire substantiel, ce qui constitue un avantage appréciable lorsqu'il s'agit de dispositifs, telle notamment une carte à puce, dont les ressources de stockage sont limitées par nature.

Selon une deuxième variante de réalisation du procédé selon l'invention, il est également possible d'utiliser les éléments d'information de type pour indiquer si un objet est encore utilisé (c'est-à-dire doit être conservé) ou peut être effacé de la "*zone variable locale*". En effet, au bout d'un certain nombre d'opérations, un objet donné enregistré dans cette zone n'est plus utilisé. Le laisser en permanence constitue donc une perte inutile d'espace mémoire.

A titre d'exemple non limitatif, on peut ajouter un bit d'information aux codes enregistrés dans les zones de "*Typage*", faisant fonction de drapeau, ou "*flag*" selon la terminologie anglo-saxonne. L'état de ce bit indique alors si l'objet doit être conservé (car encore utilisé) ou peut être effacé, et le marque comme tel. Les conventions arbitraires suivantes peuvent être adoptes :

- état logique "0" = objet utilisé
- état logique "1" = objet pouvant être effacé

Cette disposition, que l'on peut qualifier de mécanisme de type "garbage collector" (ou "ramasse-miettes") permet aussi un gain en espace mémoire.

Naturellement, les dispositions propres aux deux variantes de réalisation supplémentaires qui viennent d'être décrites peuvent être cumulées.

La figure 3 illustre schématiquement un exemple d'architecture de système informatique à base d'applications de carte à puce pour la mise en œuvre du procédé selon l'invention qui vient d'être décrit.

Ce système comprend un terminal 7, qui peut être relié ou non à des réseaux extérieurs, notamment au réseau Internet *RI*, par un modem ou tous moyens équivalents 71. Le terminal 7, par exemple un micro-ordinateur, comprend notamment un compilateur 9. Le code peut être compilé à l'extérieur du terminal pour donner un fichier dit " Class" (compilateur "JAVA" vers "Class"), c'est ce fichier qui est téléchargé par un navigateur Internet, le micro-ordinateur comprend lui un convertisseurr qui donne un fichier dit "Cap" ("Class" vers "Cap"). Ce convertisseur réduit notamment la taille du fichier "Class" pour permettre de le charger sur une carte à puce. Une application quelconque, par exemple téléchargée via le réseau Internet *RI* et écrite en langage "JAVA" est compilée par le compilateur 9 et chargée, via un lecteur de carte à puce 70 dans les circuits de mémoire 1 de la carte à puce 8. Celle-ci intègre, comme il a été rappelé, une machine virtuelle "JAVA (JVM) 6 capable d'interpréter le "p-code" issu de la compilation et

chargés dans la mémoire 1. On a également représenté différentes piles de mémoire : les zones "*zone data*" 2 et "*zone variable locale*" 3, ainsi que les zones de typage, 4 et 5, ces dernières spécifiques à l'invention. La carte à puce 8 comprend également des moyens classiques de traitement de données reliés à la mémoire 1, par exemple un microprocesseur 80.

Les communications entre la carte à puce 8 et le terminal 7, via le lecteur 70, d'une part, et entre le terminal 7 et le monde extérieur, par exemple le réseau Internet *RI*, via le modem 71, d'autre part, s'effectuent de façon également classique en soi, et il n'y pas lieu de les décrire plus avant.

A la lecture de ce qui précède, on constate aisément que l'invention atteint bien les buts qu'elle s'est fixés.

Elle permet une exécution sécurisée d'une suite d'instructions d'une application écrite langage du type à données typées se déroulant dans une mémoire à architecture de type pile. Le degré de sécurisation élevé est obtenu notamment du fait que la vérification du code est effectuée de façon dynamique, selon un des aspects de l'invention.

Cette disposition permet en outre, au prix d'une augmentation minime du temps de traitement, de se passer d'un vérificateur nécessitant des ressources de mémoire importantes. Ce type de vérificateur ne peut d'ailleurs convenir, dans la pratique, aux applications préférées de l'invention.

Il doit être clair cependant que l'invention n'est pas limitée aux seuls exemples de réalisations explicitement décrits, notamment en relation avec les figures 1A à 1G, 2A à 2B et 3.

De même, bien que l'invention s'applique plus particulièrement à un langage de type objet, et plus particulièrement au "p-code" du langage "JAVA", obtenu après compilation, elle s'applique à un grand nombre de langage mettant en œuvre des données typées, tels les langages "ADA" ou "KAMEL" rappelés dans le préambule de la présente description.

Enfin, bien que l'invention soit particulièrement avantageuse pour des systèmes embarqués à puce électronique, dont les ressources

informatiques, tant de traitement de données que de stockage de ces données, sont limitées, notamment pour des cartes à puce, elle convient parfaitement, *a fortiori*, pour des systèmes plus puissants.

TABLE I

Préfixe	Type	Code
i	"*Int*"	000
s	"*Short*"	001
b	"*Byte*"	010
z	"*Boolean*"	011
a	"*Object Reference*"	100

REVENDICATIONS

1. Procédé pour l'exécution sécurisée d'une séquence d'instructions d'une application informatique se présentant sous la forme de données typées enregistrées dans une première série d'emplacements déterminés d'une mémoire d'un système informatique, notamment un système embarqué à puce électronique, caractérisé en ce que des données supplémentaires dites éléments d'information de type sont associées à chacune desdites données typées, de manière à spécifier le type de ces données, en ce que lesdits éléments d'information de type sont enregistrés dans une deuxième série d'emplacements de mémoire déterminés (4, 5) de ladite mémoire (1) de système informatique (8), et en ce que, avant l'exécution d'instructions d'un type prédéterminé, il est procédé à une vérification en continu, préalable à l'exécution d'instructions prédéterminées, de la concordance entre un type indiqué par ces instructions et un type attendu indiqué par lesdits éléments d'information de type enregistrés dans ladite deuxième série d'emplacement de mémoire (4, 5), de manière n'autoriser ladite exécution qu'en cas de concordance entre lesdits types.

2. Procédé selon la revendication 1, caractérisé en ce que chacun desdits éléments d'information de type est constitué par une suite de bits enregistrés dans des emplacements de mémoire de ladite deuxième série (4, 5), en correspondance biunivoque avec des emplacements de mémoire de ladite première série (2, 3) dans lesquels sont enregistrées desdites données typées associées, et dont la configuration est représentative d'un desdits types de données typées.

3. Procédé selon la revendication 1, caractérisé en ce que lesdites instructions étant celles d'une application écrite en langage "JAVA" (marque déposée), lesdites données typées sont constituées par des

objets typés, en ce que ledit système informatique intègre une pièce de logicielle dite machine virtuelle "JAVA" (5) manipulant lesdits objets typés, en ce que, lesdits emplacements de mémoire (2-5) de ladite mémoire (1) du système informatique (8) étant organisés en piles comportant un nombre maximum de niveaux déterminé, chaque niveau constituant un desdits emplacements de mémoire, lesdits objets typés sont enregistrés dans au moins une première pile élémentaire dite zone de données (2) et un deuxième pile élémentaire dite zone de variables locales (3), et en ce que lesdits éléments d'information de type sont répartis dans deux piles élémentaires supplémentaires (4, 5) en relation biunivoque avec lesdites première (2) et deuxième (3) piles élémentaires, de manière à spécifier le type desdits objets associés enregistrés dans lesdites zones de données (2) et de variables locales (3).

4. Procédé selon la revendication 1, caractérisé en ce que lorsque ladite concordance n'est pas réalisée, l'exécution de ladite séquence d'instructions est interrompue et remplacée par l'exécution d'instructions correspondant à des mesures sécuritaires pré-programmées .

5. Procédé selon la revendication 3, caractérisé en ce que lesdits éléments d'information de type sont associés à des éléments d'information supplémentaires déterminant la taille desdits emplacements de mémoires desdites piles (2, 3) enregistrant lesdits objets typés, de manière à rendre variable la taille desdites piles, en fonction desdits objets à manipuler.

6. Procédé selon la revendication 3, caractérisé en ce que lesdits éléments d'information de type sont associés à des éléments d'information supplémentaires, dits drapeaux, de manière à marquer lesdits objets qui leur sont associés et à indiquer s'ils doivent être conservés dans lesdites piles (2, 3) ou peuvent être effacés.

7. Système embarqué à carte à puce électronique comprenant des moyens de traitement informatique de données et des moyens de mémoire pour l'exécution sécurisée d'une séquence d'instructions d'une application informatique se présentant sous la forme de données typées enregistrées dans une première série d'emplacements déterminés d'une mémoire d'un système informatique, caractérisé en ce que lesdits moyens de mémoire (1) comprennent une deuxième série d'emplacements déterminés (4, 5) pour l'enregistrement de données supplémentaires dites éléments d'information de type, associés à chacune desdites données typées, de manière à spécifier le type de ces données, et des moyens de vérification (6) permettant une vérification en continu, préalable à l'exécution d'instructions prédéterminées, de la concordance entre un type indiqué par ces instructions et un type indiqué par lesdits éléments d'information de type, de manière n'autoriser ladite exécution qu'en cas de concordance entre lesdits types.

8. Système selon la revendication 7, caractérisé en ce que, ladite première série d'emplacements déterminés de ladite mémoire (1) du système embarqué à puce électronique (8) étant organisée en piles comportant un nombre maximum de niveaux déterminé, chaque niveau constituant un desdits emplacements de mémoire, lesdites données typées sont enregistrées dans au moins une première pile élémentaire dite zone de données (2) et une deuxième pile élémentaire dite zone de variables locales (3), et en ce que ladite deuxième série d'emplacements de mémoire est aussi organisée en piles élémentaires (4, 5), en relation biunivoque avec lesdites première (2) et deuxième (3) piles élémentaires.

9. Système selon la revendication 8, caractérisé en ce que lesdits éléments d'information de type enregistrés dans ladite deuxième série d'emplacements de mémoire (4, 5) sont associés à des éléments d'information supplémentaires déterminant la taille desdits emplacements de mémoires desdites piles (2, 3) enregistrant lesdites données typées.

Système selon la revendication 7, caractérisé en ce que ledit système embarqué est une carte à puce (8).

This Page Blank (uspto)

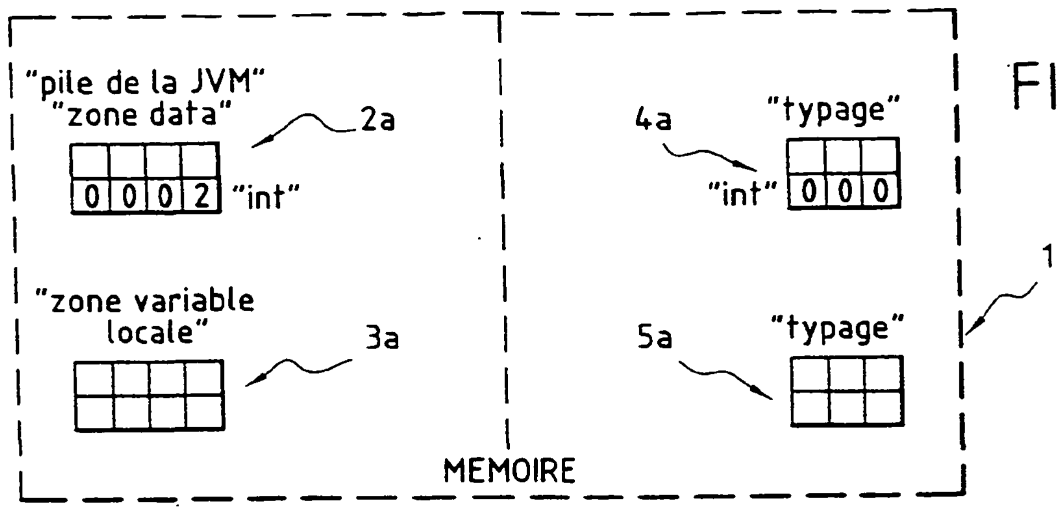


FIG.1A

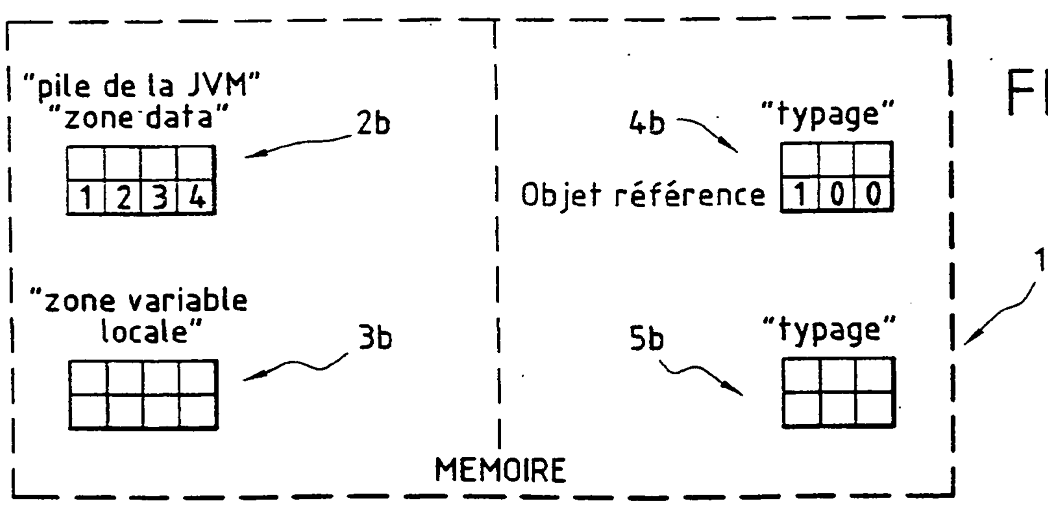


FIG.1B

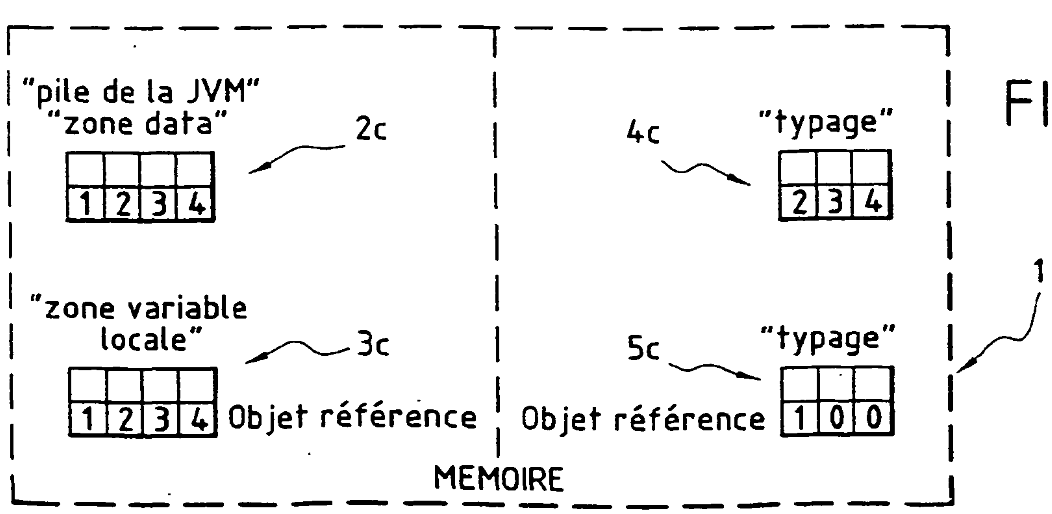


FIG.1C

This Page Blank (uspto)

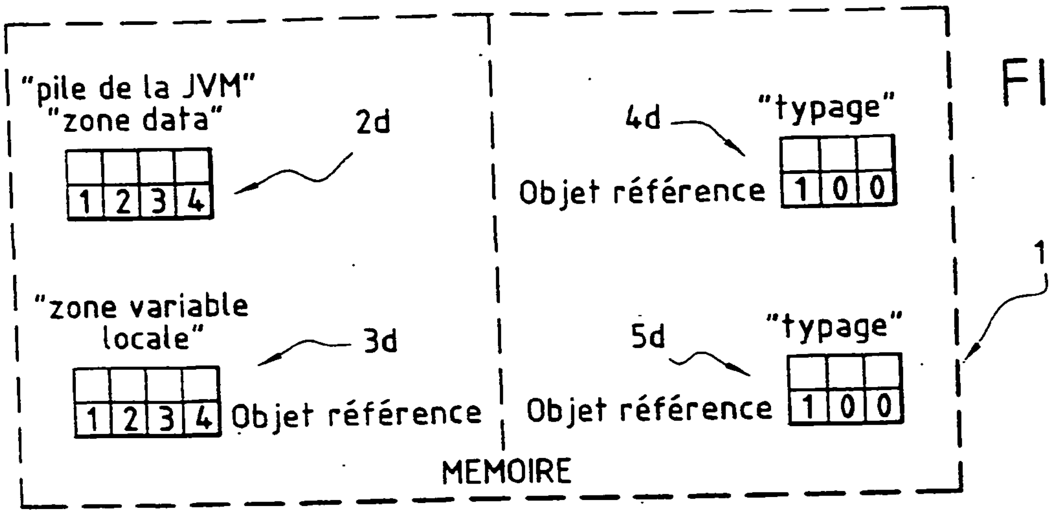


FIG.1D

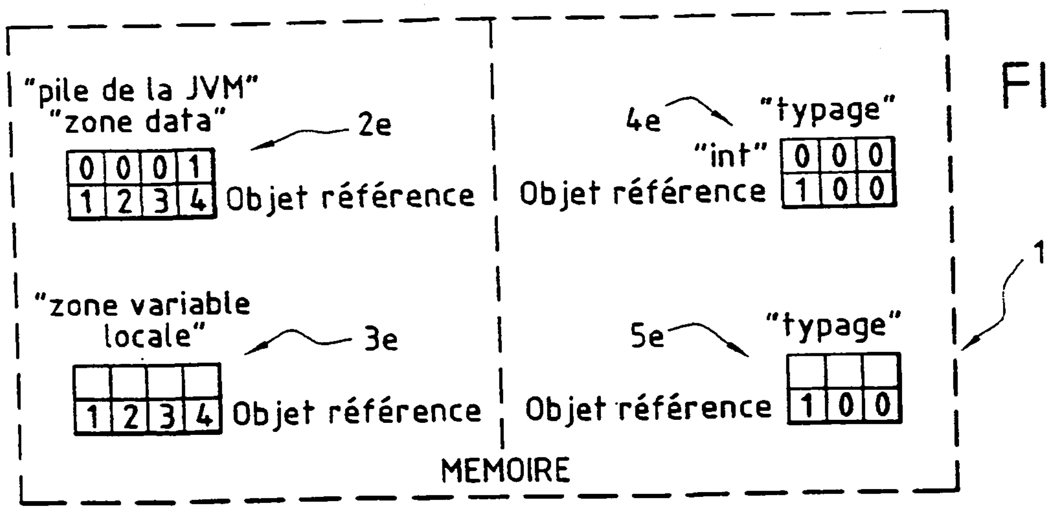


FIG.1E

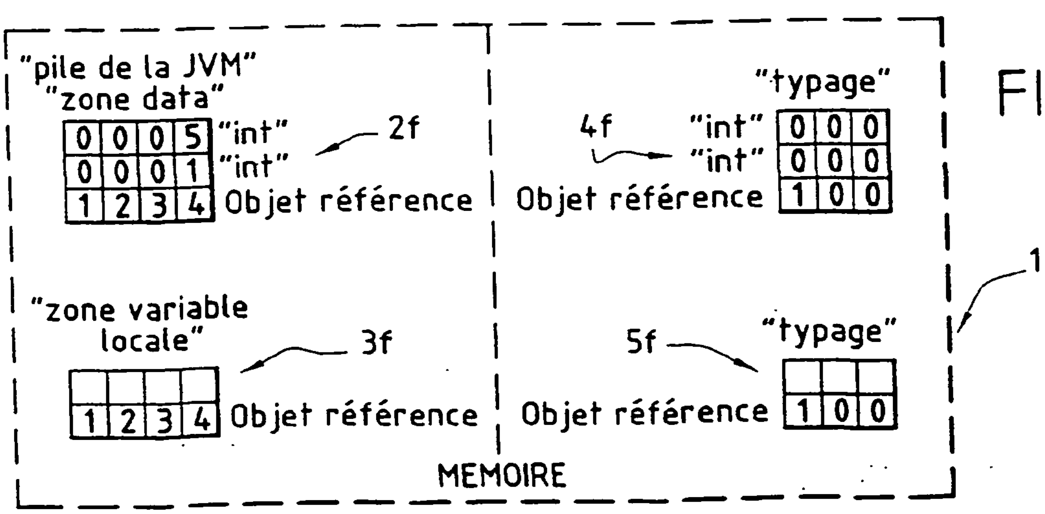


FIG.1F

This Page Blank (uspto)

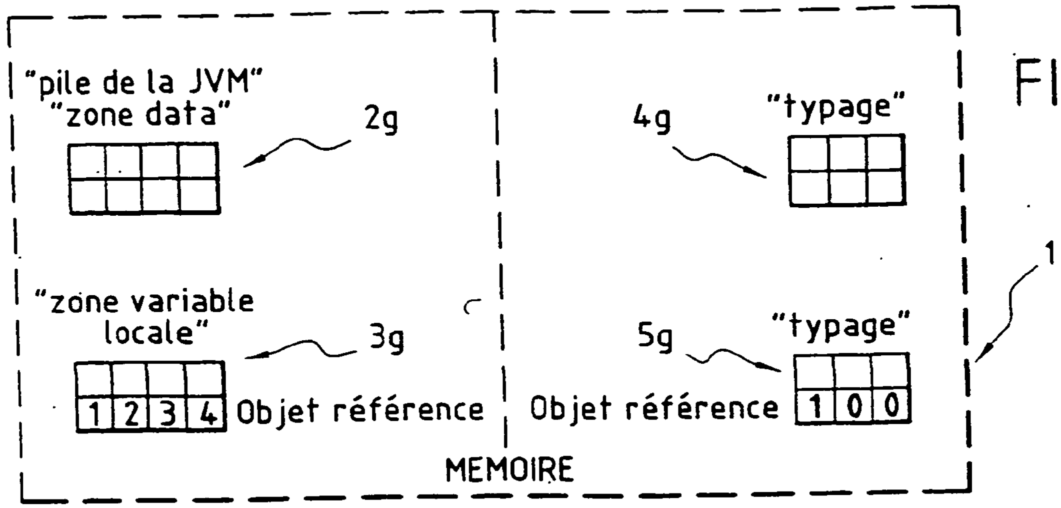


FIG.1G

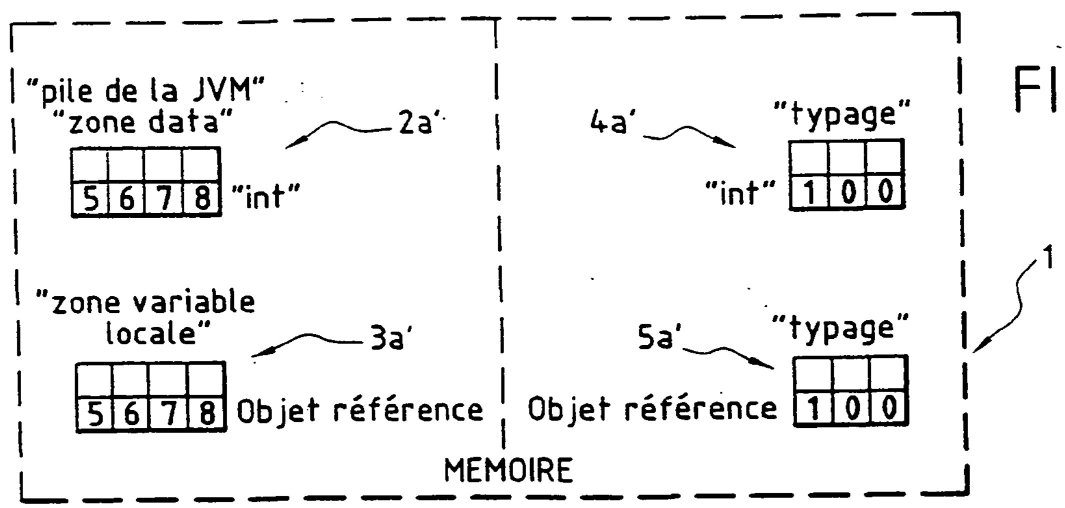


FIG.2A

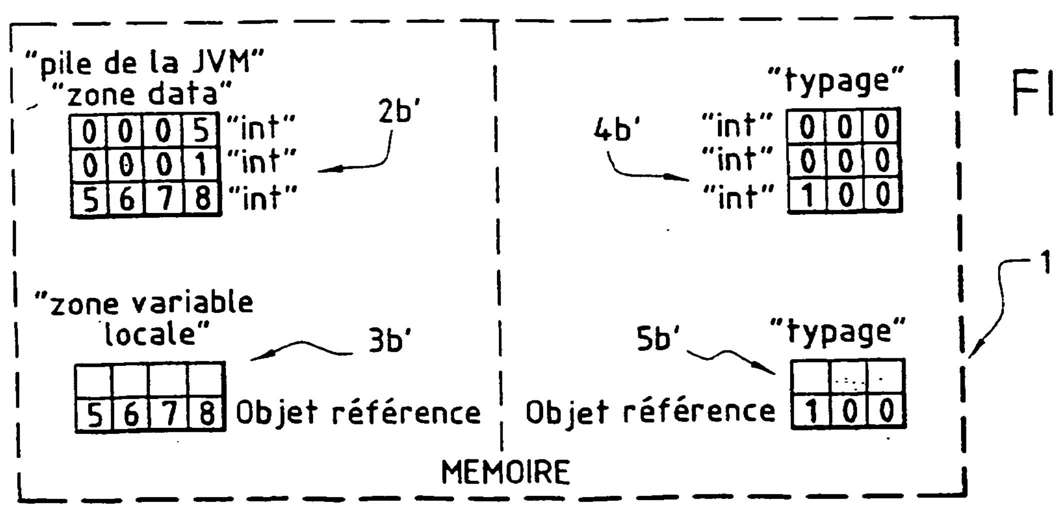


FIG.2B

This Page Blank (uspto)

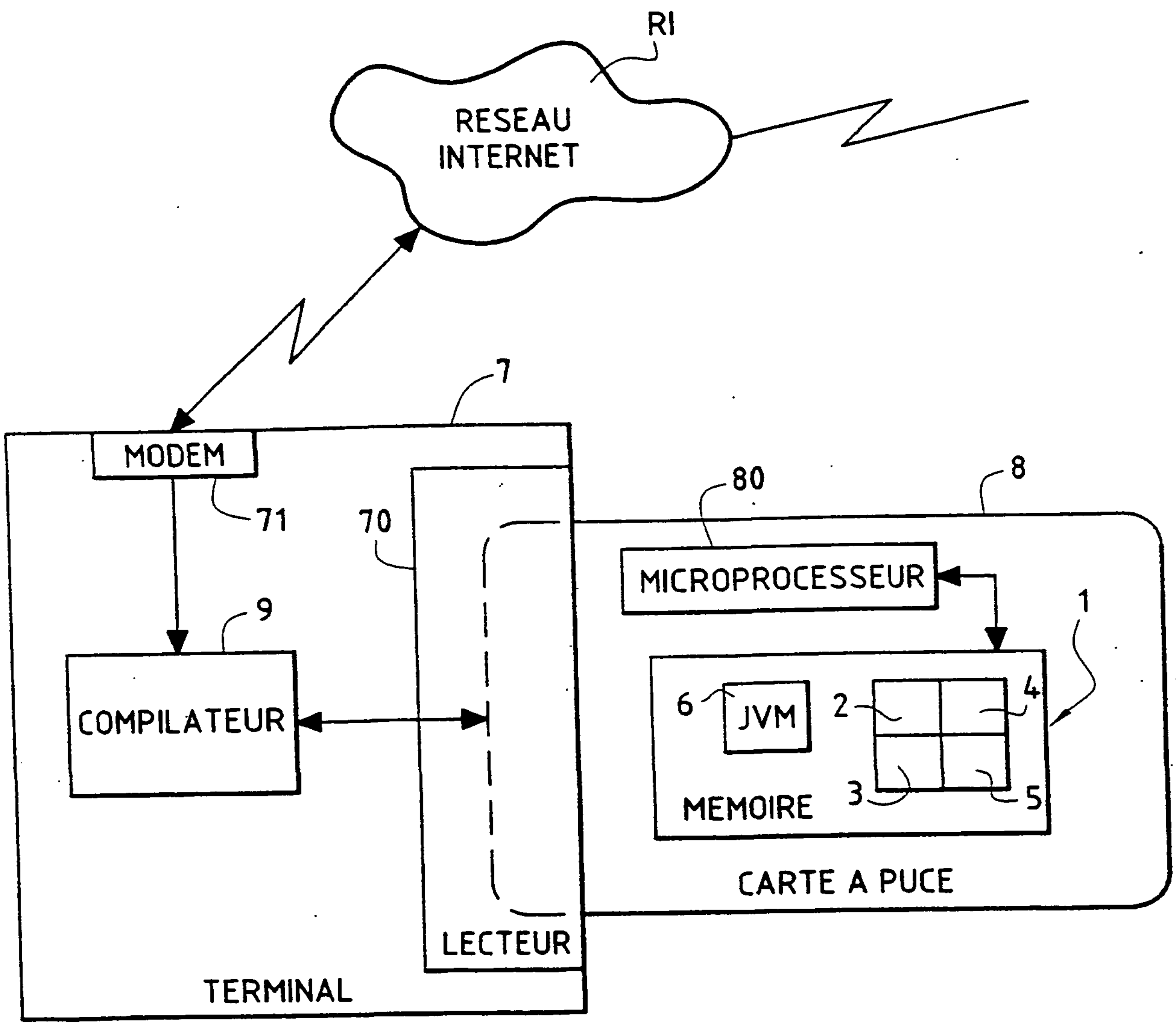

RI
RESEAU
INTERNET
7
MODEM
71
70
80
8
MICROPROCESSEUR
9
COMPILATEUR
1
6
JVM
2
4
3
5
MEMOIRE
CARTE A PUCE
LECTEUR
TERMINAL

FIG.3

This Page Blank (uspto)

INTERNATIONAL SEARCH REPORT

International Application No
PCT/FR 01/01506

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/455 G06F9/445

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, INSPEC, IBM-TDB

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	STEENKISTE P ET AL: "TAGS AND TYPE CHECKING IN LISP: HARDWARE AND SOFTWARE APPROACHES" OPERATING SYSTEMS REVIEW (SIGOPS),US,ACM HEADQUARTER. NEW YORK, vol. 21, no. 4, 1 October 1987 (1987-10-01), pages 50-59, XP000001708	1,2,4
Y	the whole document	6
A		7

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
25 July 2001	02/08/2001

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Bijn, K

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
Y	HEONSHIK SHIN ET AL: "Concurrent garbage collection with associative tag" SECOND INTERNATIONAL CONFERENCE ON COMPUTERS AND APPLICATIONS (CAT. NO.87CH2433-1), BEIJING, CHINA, 23-27 JUNE 1987, pages 230-236, XP002161891 1987, Washington, DC, USA, IEEE Comput. Soc. Press, USA ISBN: 0-8186-0780-7 page 231 -page 232, paragraph II	6
X	GRIMAUD G ET AL: "FACADE: a typed intermediate language dedicated to smart cards" ESEC/FSE'99. 7TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE. HELD JOINTLY WITH 7TH ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, TOULOUSE, FRANCE, 6-10 SEPT. 1999, vol. 24, no. 6, pages 476-493, XP002161892 Software Engineering Notes, Nov. 1999, ACM, USA ISSN: 0163-5948	7,10
A	page 480, line 14 - last line	1,3
X	COHEN, RICHARD: "Defensive Java Virtual Machine" VERSION 0.5 ALPHA RELEASE, 'Online! 13 May 1997 (1997-05-13), page 1-14 23-38 61-62 93-94 131-133 XP002161893 Austin, Texas (USA) Retrieved from the Internet: <URL:http://www.cli.com/software/djvm/index.html> 'retrieved on 2001-03-02!	1
A	the whole document	3,7
A	MCGRAW G ET AL: "JAVA SECURITY AND TYPE SAFETY" BYTE,US,MCGRAW-HILL INC. ST PETERBOROUGH, vol. 22, no. 1, 1997, pages 63-64, XP000679974 ISSN: 0360-5280 page 64, middle column, line 4 - line 12	1-17
A	EP 0 718 764 A (SUN MICROSYSTEMS INC) 26 June 1996 (1996-06-26) page 2, line 33 -page 3, line 7	1-10

INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No
PCT/FR 01/01506

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 0718764 A	26-06-1996	US 5748964 A	05-05-1998
		EP 1118940 A	25-07-2001
		JP 8234994 A	13-09-1996
		US 5999731 A	07-12-1999
		US 6247171 B	12-06-2001
		US 5740441 A	14-04-1998

This Page Blank (uspto)

RAPPORT DE RECHERCHE INTERNATIONALE

Demande Internationale No
PCT/FR 01/01506

A. CLASSEMENT DE L'OBJET DE LA DEMANDE
CIB 7 G06F9/455 G06F9/445

Selon la classification internationale des brevets (CIB) ou à la fois selon la classification nationale et la CIB

B. DOMAINES SUR LESQUELS LA RECHERCHE A PORTE

Documentation minimale consultée (système de classification suivi des symboles de classement)
CIB 7 G06F

Documentation consultée autre que la documentation minimale dans la mesure où ces documents relèvent des domaines sur lesquels a porté la recherche

Base de données électronique consultée au cours de la recherche internationale (nom de la base de données, et si réalisable, termes de recherche utilisés)

EPO-Internal, INSPEC, IBM-TDB

C. DOCUMENTS CONSIDERES COMME PERTINENTS

Catégorie °	Identification des documents cités, avec, le cas échéant, l'indication des passages pertinents	no. des revendications visées
X	STEENKISTE P ET AL: "TAGS AND TYPE CHECKING IN LISP: HARDWARE AND SOFTWARE APPROACHES" OPERATING SYSTEMS REVIEW (SIGOPS),US,ACM HEADQUARTER. NEW YORK, vol. 21, no. 4, 1 octobre 1987 (1987-10-01), pages 50-59, XP000001708	1,2,4
Y	le document en entier	6
A		7

	-/--	

[X] Voir la suite du cadre C pour la fin de la liste des documents

[X] Les documents de familles de brevets sont indiqués en annexe

° Catégories spéciales de documents cités:

"A" document définissant l'état général de la technique, non considéré comme particulièrement pertinent

"E" document antérieur, mais publié à la date de dépôt international ou après cette date

"L" document pouvant jeter un doute sur une revendication de priorité ou cité pour déterminer la date de publication d'une autre citation ou pour une raison spéciale (telle qu'indiquée)

"O" document se référant à une divulgation orale, à un usage, à une exposition ou tous autres moyens

"P" document publié avant la date de dépôt international, mais postérieurement à la date de priorité revendiquée

"T" document ultérieur publié après la date de dépôt international ou la date de priorité et n'appartenenant pas à l'état de la technique pertinent, mais cité pour comprendre le principe ou la théorie constituant la base de l'invention

"X" document particulièrement pertinent; l'inven tion revendiquée ne peut être considérée comme nouvelle ou comme impliquant une activité inventive par rapport au document considéré isolément

"Y" document particulièrement pertinent; l'inven tion revendiquée ne peut être considérée comme impliquant une activité inventive lorsque le document est associé à un ou plusieurs autres documents de même nature, cette combinaison étant évidente pour une personne du métier

"&" document qui fait partie de la même famille de brevets

Date à laquelle la recherche internationale a été effectivement achevée	Date d'expédition du présent rapport de recherche internationale
25 juillet 2001	02/08/2001

Nom et adresse postale de l'administration chargée de la recherche internationale	Fonctionnaire autorisé
Office Européen des Brevets, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Bijn, K

Formulaire PCT/ISA/210 (deuxième feuille) (juillet 1992)

C.(suite) DOCUMENTS CONSIDERES COMME PERTINENTS

Catégorie °	Identification des documents cités, avec,le cas échéant, l'indicationdes passages pertinents	no. des revendications visées
Y	HEONSHIK SHIN ET AL: "Concurrent garbage collection with associative tag" SECOND INTERNATIONAL CONFERENCE ON COMPUTERS AND APPLICATIONS (CAT. NO.87CH2433-1), BEIJING, CHINA, 23-27 JUNE 1987, pages 230-236, XP002161891 1987, Washington, DC, USA, IEEE Comput. Soc. Press, USA ISBN: 0-8186-0780-7 page 231 -page 232, alinéa II ---	6
X	GRIMAUD G ET AL: "FACADE: a typed intermediate language dedicated to smart cards" ESEC/FSE'99. 7TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE. HELD JOINTLY WITH 7TH ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, TOULOUSE, FRANCE, 6-10 SEPT. 1999, vol. 24, no. 6, pages 476-493, XP002161892 Software Engineering Notes, Nov. 1999, ACM, USA ISSN: 0163-5948	7,10
A	page 480, ligne 14 - dernière ligne ---	1,3
X	COHEN, RICHARD: "Defensive Java Virtual Machine" VERSION 0.5 ALPHA RELEASE, 'en ligne! 13 mai 1997 (1997-05-13), page 1-14 23-38 61-62 93-94 131-133 XP002161893 Austin, Texas (USA) Extrait de l'Internet: <URL:http://www.cli.com/software/djvm/index.html> 'extrait le 2001-03-02!	1
A	le document en entier ---	3,7
A	MCGRAW G ET AL: "JAVA SECURITY AND TYPE SAFETY" BYTE,US,MCGRAW-HILL INC. ST PETERBOROUGH, vol. 22, no. 1, 1997, pages 63-64, XP000679974 ISSN: 0360-5280 page 64, colonne du milieu, ligne 4 - ligne 12 ---	1-17
A	EP 0 718 764 A (SUN MICROSYSTEMS INC) 26 juin 1996 (1996-06-26) page 2, ligne 33 -page 3, ligne 7 -----	1-10

RAPPORT DE RECHERCHE INTERNATIONALE

Renseignements relatifs aux membres de familles de brevets

Demande Internationale No
PCT/FR 01/01506

Document brevet cité au rapport de recherche	Date de publication	Membre(s) de la famille de brevet(s)	Date de publication
EP 0718764 A	26-06-1996	US 5748964 A	05-05-1998
		EP 1118940 A	25-07-2001
		JP 8234994 A	13-09-1996
		US 5999731 A	07-12-1999
		US 6247171 B	12-06-2001
		US 5740441 A	14-04-1998

This Page Blank (uspto)

TRAITE DE COOPERATION EN MATIERE DE BREVETS

PCT

RAPPORT DE RECHERCHE INTERNATIONALE

(article 18 et règles 43 et 44 du PCT)

Référence du dossier du déposant ou du mandataire PCT 3884/BC	**POUR SUITE A DONNER** voir la notification de transmission du rapport de recherche internationale (formulaire PCT/ISA/220) et, le cas échéant, le point 5 ci-après	
Demande internationale n° PCT/FR 01/ 01506	Date du dépôt international *(jour/mois/année)* 17/05/2001	(Date de priorité (la plus ancienne) *(jour/mois/année)* 17/05/2000
Déposant BULL CP8		

Le présent rapport de recherche internationale, établi par l'administration chargée de la recherche internationale, est transmis au déposant conformément à l'article 18. Une copie en est transmise au Bureau international.

Ce rapport de recherche internationale comprend 3 feuilles.

[X] Il est aussi accompagné d'une copie de chaque document relatif à l'état de la technique qui y est cité.

1. **Base du rapport**

a. En ce qui concerne la **langue**, la recherche internationale a été effectuée sur la base de la demande internationale dans la langue dans laquelle elle a été déposée, sauf indication contraire donnée sous le même point.

[ ] la recherche internationale a été effectuée sur la base d'une traduction de la demande internationale remise à l'administration.

b. En ce qui concerne **les séquences de nucléotides ou d'acides aminés** divulguées dans la demande internationale (le cas échéant), la recherche internationale a été effectuée sur la base du listage des séquences :

[ ] contenu dans la demande internationale, sous forme écrite.
[ ] déposée avec la demande internationale, sous forme déchiffrable par ordinateur.
[ ] remis ultérieurement à l'administration, sous forme écrite.
[ ] remis ultérieurement à l'administration, sous forme déchiffrable par ordinateur.
[ ] La déclaration, selon laquelle le listage des séquences présenté par écrit et fourni ultérieurement ne vas pas au-delà de la divulgation faite dans la demande telle que déposée, a été fournie.
[ ] La déclaration, selon laquelle les informations enregistrées sous forme déchiffrable par ordinateur sont identiques à celles du listage des séquences présenté par écrit, a été fournie.

2. [ ] **Il a été estimé que certaines revendications ne pouvaient pas faire l'objet d'une recherche** (voir le cadre I).

3. [ ] **Il y a absence d'unité de l'invention** (voir le cadre II).

4. En ce qui concerne le **titre,**

[X] le texte est approuvé tel qu'il a été remis par le déposant.
[ ] Le texte a été établi par l'administration et a la teneur suivante:

5. En ce qui concerne **l'abrégé,**

[X] le texte est approuvé tel qu'il a été remis par le déposant
[ ] le texte (reproduit dans le cadre III) a été établi par l'administration conformément à la règle 38.2b). Le déposant peut présenter des observations à l'administration dans un délai d'un mois à compter de la date d'expédition du présent rapport de recherche internationale.

6. La figure **des dessins** à publier avec l'abrégé est la Figure n° 3

[X] suggérée par le déposant.
[ ] parce que le déposant n'a pas suggéré de figure.
[ ] parce que cette figure caractérise mieux l'invention.

[ ] Aucune des figures n'est à publier.

This Page Blank (uspto)

RAPPORT DE RECHERCHE INTERNATIONALE

Demande Internationale No
PCT/FR 01/01506

A. CLASSEMENT DE L'OBJET DE LA DEMANDE
CIB 7 G06F9/455 G06F9/445

Selon la classification internationale des brevets (CIB) ou à la fois selon la classification nationale et la CIB

B. DOMAINES SUR LESQUELS LA RECHERCHE A PORTE

Documentation minimale consultée (système de classification suivi des symboles de classement)
CIB 7 G06F

Documentation consultée autre que la documentation minimale dans la mesure où ces documents relèvent des domaines sur lesquels a porté la recherche

Base de données électronique consultée au cours de la recherche internationale (nom de la base de données, et si réalisable, termes de recherche utilisés)

EPO-Internal, INSPEC, IBM-TDB

C. DOCUMENTS CONSIDERES COMME PERTINENTS

Catégorie °	Identification des documents cités, avec, le cas échéant, l'indication des passages pertinents	no. des revendications visées
X	STEENKISTE P ET AL: "TAGS AND TYPE CHECKING IN LISP: HARDWARE AND SOFTWARE APPROACHES" OPERATING SYSTEMS REVIEW (SIGOPS),US,ACM HEADQUARTER. NEW YORK, vol. 21, no. 4, 1 octobre 1987 (1987-10-01), pages 50-59, XP000001708	1,2,4
Y	le document en entier	6
A		7

	-/--	

[X] Voir la suite du cadre C pour la fin de la liste des documents

[X] Les documents de familles de brevets sont indiqués en annexe

° Catégories spéciales de documents cités:

"A" document définissant l'état général de la technique, non considéré comme particulièrement pertinent

"E" document antérieur, mais publié à la date de dépôt international ou après cette date

"L" document pouvant jeter un doute sur une revendication de priorité ou cité pour déterminer la date de publication d'une autre citation ou pour une raison spéciale (telle qu'indiquée)

"O" document se référant à une divulgation orale, à un usage, à une exposition ou tous autres moyens

"P" document publié avant la date de dépôt international, mais postérieurement à la date de priorité revendiquée

"T" document ultérieur publié après la date de dépôt international ou la date de priorité et n'appartenenant pas à l'état de la technique pertinent, mais cité pour comprendre le principe ou la théorie constituant la base de l'invention

"X" document particulièrement pertinent; l'inven tion revendiquée ne peut être considérée comme nouvelle ou comme impliquant une activité inventive par rapport au document considéré isolément

"Y" document particulièrement pertinent; l'inven tion revendiquée ne peut être considérée comme impliquant une activité inventive lorsque le document est associé à un ou plusieurs autres documents de même nature, cette combinaison étant évidente pour une personne du métier

"&" document qui fait partie de la même famille de brevets

Date à laquelle la recherche internationale a été effectivement achevée	Date d'expédition du présent rapport de recherche internationale
25 juillet 2001	02/08/2001
Nom et adresse postale de l'administration chargée de la recherche internationale Office Européen des Brevets, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Fonctionnaire autorisé Bijn, K

1

Formulaire PCT/ISA/210 (deuxième feuille) (juillet 1992)

This Page Blank (uspto)

RAPPORT DE RECHERCHE INTERNATIONALE

Demande Internationale No
PCT/FR 01/01506

C.(suite) DOCUMENTS CONSIDERES COMME PERTINENTS

Catégorie °	Identification des documents cités, avec,le cas échéant, l'indicationdes passages pertinents	no. des revendications visées
Y	HEONSHIK SHIN ET AL: "Concurrent garbage collection with associative tag" SECOND INTERNATIONAL CONFERENCE ON COMPUTERS AND APPLICATIONS (CAT. NO.87CH2433-1), BEIJING, CHINA, 23-27 JUNE 1987, pages 230-236, XP002161891 1987, Washington, DC, USA, IEEE Comput. Soc. Press, USA ISBN: 0-8186-0780-7 page 231 -page 232, alinéa II ---	6
X	GRIMAUD G ET AL: "FACADE: a typed intermediate language dedicated to smart cards" ESEC/FSE'99. 7TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE. HELD JOINTLY WITH 7TH ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, TOULOUSE, FRANCE, 6-10 SEPT. 1999, vol. 24, no. 6, pages 476-493, XP002161892 Software Engineering Notes, Nov. 1999, ACM, USA ISSN: 0163-5948	7,10
A	page 480, ligne 14 - dernière ligne ---	1,3
X	COHEN, RICHARD: "Defensive Java Virtual Machine" VERSION 0.5 ALPHA RELEASE, 'en ligne! 13 mai 1997 (1997-05-13), page 1-14 23-38 61-62 93-94 131-133 XP002161893 Austin, Texas (USA) Extrait de l'Internet: <URL:http://www.cli.com/software/djvm/index.html> 'extrait le 2001-03-02!	1
A	le document en entier ---	3,7
A	MCGRAW G ET AL: "JAVA SECURITY AND TYPE SAFETY" BYTE,US,MCGRAW-HILL INC. ST PETERBOROUGH, vol. 22, no. 1, 1997, pages 63-64, XP000679974 ISSN: 0360-5280 page 64, colonne du milieu, ligne 4 - ligne 12 ---	1-17
A	EP 0 718 764 A (SUN MICROSYSTEMS INC) 26 juin 1996 (1996-06-26) page 2, ligne 33 -page 3, ligne 7 -----	1-10

1

This Page Blank (uspto)

RAPPORT DE RECHERCHE INTERNATIONALE

Renseignements relatifs aux membres de familles de brevets

Demande Internationale No
PCT/FR 01/01506

Document brevet cité au rapport de recherche	Date de publication	Membre(s) de la famille de brevet(s)	Date de publication
EP 0718764 A	26-06-1996	US 5748964 A	05-05-1998
		EP 1118940 A	25-07-2001
		JP 8234994 A	13-09-1996
		US 5999731 A	07-12-1999
		US 6247171 B	12-06-2001
		US 5740441 A	14-04-1998

This Page Blank (uspto)

83 Operating Systems Review
21. (1987)Oct., No.4, Washington,DC,USA

E
XP 000001708
p 50-59
G06F 9/44K
G06F 11/08

Tags and Type Checking in LISP: Hardware and Software Approaches

Peter Steenkiste and John Hennessy

Computer Systems Laboratory
Stanford University

Abstract

One of the major factors that distinguishes LISP from many other languages (Pascal, C, Fortran, etc.) is the need for run-time type checking. Run-time type checking is implemented by adding to each data object a tag that encodes type information. Tags must be compared for type compatibility, removed when using the data, and inserted when new data items are created. This tag manipulation, together with other work related to dynamic type checking and generic operations, constitutes a significant component of the execution time of LISP programs. This has led both to the development of LISP machines that support tag checking in hardware and to the avoidance of type checking by users running on stock hardware. To understand the role and necessity of special-purpose hardware for tag handling, we first measure the cost of type checking operations for a group of LISP programs. We then examine hardware and software implementations of tag operations and estimate the cost of tag handling with the different tag implementation schemes. The data shows that minimal levels of support provide most of the benefits, and that tag operations can be relatively inexpensive, even when no special hardware support is present.

1. Introduction

In statically typed languages like Pascal, type checking is done at compile-time. Languages like LISP do not require the user to specify the type of each data item so *run-time type checking* is required. Run-time type checking is implemented by adding a *tag* to each data item to encode the type of that item; operations on the data can then be type checked. On general-purpose processors, the tag is usually stored together with the data, or with a pointer to the data, in a single word. On LISP machines, the word length is often extended to accommodate tag bits, which are then handled with separate hardware. General-purpose machines must explicitly extract and compare tags when checking types and remove tags when operating on tagged data. Dynamic type checking, while chiefly concerned with tag operations, also involves support of generic arithmetic.

An earlier study [18] on the run-time behavior of LISP showed that LISP programs spend an average of one fourth of their execution time on handling tags with type checking turned off. This cost was the primary motivation for the creation of LISP machines with a *tagged architecture*. LISP machines with tag support have instructions that can operate on the tag and the data part of an item, without having to disassemble it with separate instructions, and they usually support tag checking operations in parallel with other operations. For example, an integer add and the type check on the two operands occur simultaneously. Adding run-time checking to primitive LISP operations slows down our set of LISP programs by 25%, so overlapping some or all of this testing with other operations can give a substantial speedup.

In this paper we first study the cost of tag handling for ten LISP programs that were executed on the MIPS-X reduced-instruction-set processor. We then describe a number of tag implementations, including both software schemes, which can be used on general-purpose architectures, and hardware schemes for LISP machines. We compare how effective they are at reducing the cost of tag handling; finally we discuss generic arithmetic in Section 4.

The MIPS-X research project has been supported by the Defense Advanced Research Project Agency under contract # MDA903-83-C-0335

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

2. Portable Standard Lisp on MIPS-X

The data presented in this paper are based on measurements of ten Portable Standard Lisp programs that were executed on an instruction-level simulator for MIPS-X, a high-performance microprocessor [10]. MIPS-X is used as a typical example of a reduced-

© 1987 ACM 0-89791-238-1/87/1000-0050 $00.75

BNSDOCID <XP 1708A I >

instruction-set processor. An advantage of using a RISC architecture in this type of study is that one can measure both instruction counts and execution time easily, since the latter depends directly on the former (ignoring cache misses). The programs we have studied include a compiler front-end, a garbage collector, and a rational function evaluator, and three of the larger Gabriel benchmarks [7]; together the ten programs contain about 11000 lines of LISP code, without comments. Details about the benchmarks appear in the Appendix. Portable Standard Lisp [8, 9] is a small, efficient LISP dialect. In the remainder of this section we discuss the PSL implementation on MIPS-X.

2.1. The implementation of tags

The PSL implementation on MIPS-X uses a 5 bit tag that is stored in the most significant part of the word; the remaining 27 bits contain a pointer to the data. For some data types, the data item contains immediate data, e.g. symbols and integers that fit in 27 bits. There are four operations related to tags:

- *tag insertion*: given a piece of data, or a pointer, and its type (tag value), construct the data item,
- *tag removal*: given an item, extract the data item, that is, clear the tag and create a valid pointer or data item,
- *tag extraction*: given an item, extract the tag value,
- *tag checking*: given an item, test the value of its tag; this is implemented as a tag extraction followed by a conditional branch.

The tag value for positive integers is 0, and for negative integers, 31 (all 1's). As a result of this choice, the LISP representation for an integer is the same as its two's complement machine representation [9]. This means that integer arithmetic done on short (27-bit) integers without type checking can use the arithmetic instructions of the processor without any need for reformatting. This optimization speeds up all low level integer operations. Because of the special tag encoding for integers, type checking for integers is different from other data types (see Section 4.1). Testing for overflow for integer additions (and subtractions) can be implemented as a type checking operation: if we add two LISP integer items and overflow occurs, then the result will not be a LISP integer. This special treatment of integers is justified by their high frequency of use.

2.2. Run-time checking and generic operations

How much run-time type checking is done, and how it is done, strongly influence the number of tag checking operations that are executed. For this reason, we first optimized PSL run-time checking [19] to make its performance comparable to that of some newer, optimized LISP systems [3, 11]. In this section, we describe what data types are used in our test programs, and how type checking is done for those types.

For a lot of operations, run-time checking is equivalent to checking the tag of the operand. An important example is type checking on *list operations* such as *car* and *cdr*: the operand has to be a list, otherwise the operation is illegal. Type checking for a symbol also consists of a single tag check. Run-time checking for *vector accesses* is more complicated. Compilers for language like Pascal and C often allow the programmer to specify run-time bounds checking. In LISP, vector accesses with full run-time checking will not only do bounds checking, but will also check that the indexed object is a vector and that the indexing type is legal.

Because the type of the operands of an *arithmetic operation* is not known at compile time, the LISP run-time system has to deal with type conversion and has to pick an operator that matches the type of the operands. This generic arithmetic can be implemented by doing a type dispatch on the type of the operands, but integers are by far the most common type of numbers in LISP [24], and generic arithmetic can be speeded up by first specifically testing for integer operands, thus giving a fast result for the most common integer case. The expensive general sequence is only used if non-integers are involved. The integer tests and the integer operation are done inline.

Most LISP dialects define more data types than are used in our programs, but the data objects most actively used will be of the types we discussed (numbers, symbols, lists, or vectors). A lot of the other data types are also modeled after, or are implemented on top of one of the above types, for example: *structures*, *strings*, and *bit-vectors* [17]. Because the data types used in our programs, and the implementation of tag checking are both similar to what is found in other modern LISP systems, we expect that the numbers presented in this paper are representative for most LISP dialects.

3. Time spent on tag operations

In this section we look at how much time LISP programs spend on various tag operations. LISP usually requires run-time checking on all operations, but there are several important cases where these checks are not required. First, when the compiler can determine the type of an operand based on the program context [12], or when the programmer uses variable declarations or type specific operators [16, 13, 3], the type checking operations can be removed without affecting correctness or security. Second, many LISP compilers have a flag that determines whether the compiler will give priority to speed or to safety [17]. The importance of optimizing run-time type checking

NSDOCID <XP 1708A I >

cannot be accurately measured until we know the frequency of its occurrence in real programs. Because the amount of run-time checking depends on techniques to minimize the checks, we have collected data in two extreme situations: when no type checking is done, and when full run-time checking is done. A real LISP program will lie between these two extremes.

Adding run-time checking to our set of programs increases the execution time with 25% on average, but the slowdown for individual programs ranges from 6% to 88% (Table 1). Checking on list operations is responsible for most of the increase in execution time, but for *opt* and *trav*, the contribution of checking vector operations is significant, and *rat* does a fair amount of arithmetic.

	arith	vector	list	total
inter	0.63	0.00	19.04	19.68
deduce	0.09	0.00	12.27	12.36
dedgc	0.04	0.00	6.58	6.62
rat	4.85	0.00	13.69	18.54
comp	0.05	0.00	10.34	10.39
opt	2.68	11.76	27.99	42.43
frl	0.45	0.00	9.72	10.17
boyer	0.00	0.00	17.50	17.50
brow	0.03	0.00	19.91	19.94
trav	3.09	71.96	13.19	88.25
average	1.19	8.37	15.02	24.59

Table 1: Percentage increase in execution time when run-time checking is added

In this paper, the 'cost' without (with) run-time checking, is expressed as a percentage of the execution time of the programs without (with) run-time checking. The tags are implemented as described in Section 2.1, but changes in the implementation, like putting the tags in the low order bits, should not influence our results significantly.

3.1. Tag insertion

A tag has to be inserted each time when a new item is created. Inserting a tag in a data item when both the tag and the item are in a register costs two cycles: one to shift the tag to the most significant bits, and one to 'or' the tag and the item together. Because of our choice of tag values, no tag insertion is necessary when an integer is created. Figure 1 shows that the ten programs spend on average 1.5% of their time on the insertion of tags.

We will not discuss tag insertion any further in this document, both because it is not time critical, and because there is little possibility for improvement by simple changes in software or hardware. For example, keeping a *preshifted* list tag in a register (and thus reducing the cost of tag insertion for list cells to one cycle) would speed up our programs only 0.5%.

3.2. Tag removal

On MIPS-X, the tag has to be removed before the data part of an item can be used, except for integers. Removing the tag can be done in one cycle by masking it out with a mask kept in a register. Figure 1 shows that the programs without run-time checking spend 8.7% of their time on masking out tags. With run-time checking, the cost drops to 7%, because the total execution time has increased (due to time spent on extracting and checking tags) while the time spent on tag removal stays the same.

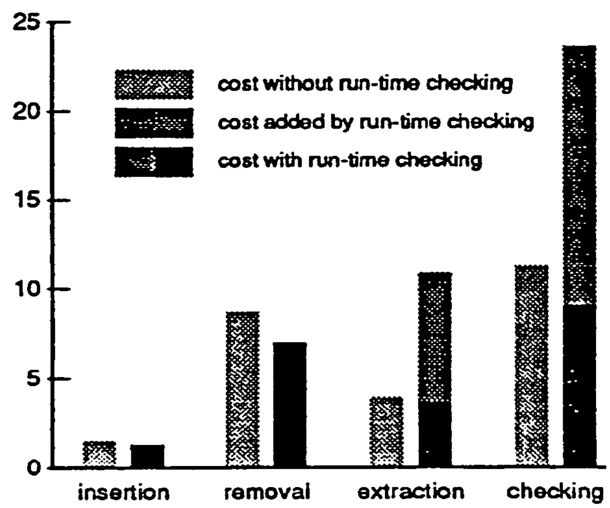


Figure 1: Percentage of time spent on all tag handling operations

3.3. Tag extraction

On MIPS-X, it is necessary to extract the tag of a data item before it can be compared with a known tag value to check the type of the item. Tag extraction can be done in a single cycle with a logical shift that places the tag in the low order part of the word. Figure 1 shows that 4% of the execution time is spent on tag extraction when no run-time checking is done. These tag extraction operations are part of type tests that are explicitly specified in the source program.

The dark histogram in Figure 1 shows the cost of tag extraction in programs with full run-time checking. The black part of the graph corresponds to the operations that were already present in the programs without run-time checking (light grey histogram), and

BNSDOCID: <XP 1703A I >

the dark grey corresponds to the operations that were added as part of the run-time checking; both components are expressed as a percentage of the execution time with run-time checking. Adding full run-time checking sharply increases the extraction cost and checking list operations is responsible for 80% of this increase. When we add run-time checking, the increase in the number of tag extraction operations, and thus in the number of tag checking operations, is about equal to the number of tag removal operations. This is what one would expect: with full run-time checking the type of each data item is checked before the item is used and using an item requires the removal of its tag.

3.4. Tag checking

Figure 1 shows that programs without run-time checking, spend 11% of their time on tag checking. The cost of tag checking includes the cost of extracting the tag, one cycle for a comparison, and possibly one or two cycles for unused branch delay slots. With full run-time checking almost 24% of the execution time is used for tag checking. Both with and without run-time testing, 95% of the tag checking operations are of the simple type (tag extraction followed by a comparison with a constant); the remaining tag checking operations are related to testing for integers and numbers.

3.5. Summary

In this section we saw that the total cost of tag handling is between 22% and 32% (Figure 1), depending on how much run-time checking is done. This cost is fairly constant across all programs - the standard deviations are 5.6% and 7.5% respectively - although the programs are widely different. In the following sections we will look at different tag implementations that reduce the cost of tag handling: in Section 4 we discuss software tag implementations that speed up integer testing, and in Sections 5 and 6 we look at schemes that can be used to reduce the cost of tag removal and tag checking. The hardware schemes proposed must be evaluated not only for improvements in instruction count, but also for potential negative impact on the processor's cycle time.

4. Software optimization of generic arithmetic and integer testing

If the tag is kept in the low order part of the word, integers should have tag value 0 for fast arithmetic, and integer testing is the same as for other data types. But when the tag is kept in the most significant part of the word, integers should get a tag that is the sign extension of their sign bit, and type checking for integers is more expensive, because positive and negative integers have different tags. This section first describes tag checking for integers, and then demonstrates how a special tag encoding can reduce the number of test-for-integer operations that are required for integer-biased generic arithmetic.

4.1. Support for integer testing

Testing for an integer, if the tag is in the most significant part of the word, can be implemented in a number of ways (assume 5 tag bits):

1. Extract the tag, then check for a positive integer; if that fails, check for a negative integer.
2. Sign extend the least significant 28 bits; the item is an integer if the result is equal to the original item.
3. Assuming that an arithmetic shift left gives a trap on overflow, doing an arithmetic shift left over 4 bits will trap if the item is not an integer.

The last method allows integer testing in a single cycle, but unfortunately, most architectures do not have a true arithmetic shift left. Recovering from a trap is also expensive, so this implementation would probably only be acceptable if a non-integer operand represents an error condition. Neither the first nor the second method require special hardware. It depends on the sign of the number which one is the fastest. The second method was used for the measurements in this paper, and it always costs 3 cycles. The first method is faster for positive numbers, and slower for negative numbers.

4.2. Reducing the cost of generic arithmetic

With integer-biased generic arithmetic (Section 2.2) the cost of generic arithmetic, averaged over the ten programs, is only 2%, but for computation intensive programs, this cost can be substantially higher. In *rat*, which is the most computation intensive program in our set, 8% of the execution time is spent on generic arithmetic. In this section we describe how the overhead of integer testing, which dominates the cost of generic arithmetic for simple integer operations, can be reduced by using a special tag encoding.

A generic integer add takes 10 cycles: 9 cycles for type and overflow checking, and 1 for adding. The reason for this high cost is that 3 type checking operations are required: two for the operands, and one to check the result for overflow. If we assign tag values in such a way that the sum of two non-integer tag values can never result in an integer tag value, then we can add the numbers, and do all the type and overflow checking with one single type checking operation on the result. This reduces the cost of a generic add of two integers to 4 cycles. For some other arithmetic operations the speedup would be smaller because more than one type test is necessary. With this special encoding of tags, the time spent on generic arithmetic drops to 1.6%, or a gain of 0.4% over the scheme with a

NSDOCID <XP 17084 I >

straightforward tag encoding; for *rat*, the speedup is about 2%.

The requirement on tag values can be met by using an extra tag bit; in the case of PSL, 6 tag bits instead of 5. With 6 tag bits it is possible to assign tag bits in such a way that the sum of two non-integer tag values, with possibly a carry in, can never result in a integer tag value without giving overflow. Another possibility is to keep 5 tag bits, and to reduce the number of tag values that are required by putting some typing information with the data.

This tag implementation has the disadvantage that it requires an extra tag bit. Not only does this reduce the address size by one bit, but it also means that this scheme cannot be used with tag implementations that allow only 2 or 3 tag bits (see Section 5.2). Since these tag implementations have a higher payoff, at least for our set of programs, we will not use the encoding described in this section. But if enough tag bits are available, and if generic arithmetic is important, then the special tag encoding deserves consideration.

5. Support for tag removal

In Section 3.2 we found that our set of LISP programs spend around 9% of their time masking out the tag of a data item in order to be able to use the data part. In this section we first discuss the need for tag removal. Then we show what can be gained if tag removal is not necessary, and finally we describe a number of tag implementations that eliminate the need for tag removal.

5.1. The need for tag removal

The data part of most LISP objects contains a pointer to the data, so it will always be used as an address. Two important exceptions are integers and symbols. We saw earlier that no tag removal is necessary for integers, and symbols are either compared with other symbols, without removing the tag, or they are used as an index in a symbol table, in which case the data part of the item is again used for the purpose of addressing a memory location. On a processor that drops the top 5 bits of 32 bit addresses before accessing memory, it is not necessary to mask the tag of an item explicitly when the data part of the item is used to access memory, which is, as we just argued, usually the case.

We changed the compiler so that no masking of the tag is done for items that are used as addresses, and we changed the simulator, so that it only uses the bottom 27 bits of an address when accessing memory. Figure 2 shows, for programs with no run-time checking, the decrease in instruction frequencies resulting from this optimization. When we compare the 'and' entry in Figure 2 with the 'removal' entry in Figure 1, we see that almost all masking operations have been removed. Part of the gain is undone by an increase in move instructions which is a consequence of the requirement that all load instructions have to be idempotent (repeatable). The increase in wasted cycles (no-ops and squashed instructions) results from the fact that fewer ALU instructions are available to fill delay slots after branches, loads and stores. Not having to mask the tag speeds up our programs 5.7% on average.

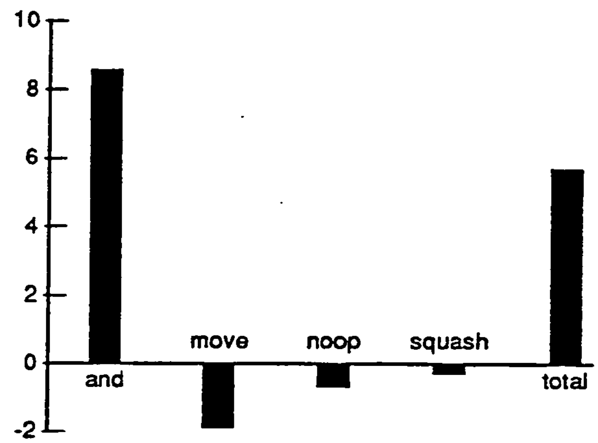


Figure 2: Reduction in instruction frequencies due to the elimination of tag removal

5.2. Implementation

Removal of the tag when an object is used as an address could be accomplished in either hardware or software. A hardware solution is available on machines where the address length is shorter than the word length and the tag can be placed in the upper bits of the word. Several architectures with this property exist (68000 [20], IBM/370 [23]), but most architectures that are designed today have a full 32 bit address space. LISP machines typically treat the tag and the data part of a data item as separate entities, so it is natural that the tag is automatically dropped when memory is accessed [1, 14].

Given a general-purpose processor like MIPS-X it would be possible to add special hardware that would blank out the 5 most significant bits of each address, before it is put on the address bus. This hardware could be controlled by a bit in the processor status word or special load and store instructions could be added to the instruction set.

It is possible to avoid the need for tag masking by making only changes in the software. On MIPS-X, most tag removals for addresses could be eliminated by using the two low-order bits of a word as a tag. MIPS-X uses byte addresses, but all memory accesses are

BNSDOCID <XP 1709A I >

word aligned, hence, the bottom two bits of an address are dropped before the (word-addressed) memory is accessed. With two tag bits, three combinations are used to encode the most frequently used data types, leaving one combination as an escape, and thus eliminating tag masking for most memory accesses. On architectures that look at the two bottom bits of the address, this approach can still be used, but the compiler has to adjust the offset that is used to access the object so that the tag is eliminated, as is done in [15].

It is possible to avoid tag removal for more LISP types by using the *three bottom bits* for tag encoding. Even and odd integers get the tag values 000 and 100, so that integer arithmetic and indexing in word vectors will be fast, four tag values can be used for frequently used data types, and the values 011 and 111 (two bottom bits 1) are reserved as an escape. For data types with 3 bit tags, data objects will always be aligned on even or odd word boundaries. This is not a problem since list cells always require two words, and other types such as vectors and structures often come in larger blocks, so wasting a word to ensure the alignment is relatively cheap. Aligning list cells on double word boundaries might also be beneficial for caches with block sizes larger than two. The Lucid Common Lisp compiler uses the three bottom bits as tag bits for some architectures.

Not having to strip the tag of an item before accessing memory can save almost 6% in execution time, and several simple implementations exist. The software schemes that place the tag in the bottom two or three bits are very attractive: they avoid tag removal for almost all memory accesses without requiring special hardware, and they have the added advantage that the address space is not limited, which is important for large LISP systems.

6. Support for tag extraction and tag checking

Our programs spend between 11% and 24% of their time on tag checking (Figure 1), depending on how much run-time checking is done, so tag checking is an attractive candidate for optimization. Tag checking operations, or type dispatching, are required in a number of situations:

1. *run-time error checking*, for example type checking as part of a *car* operation,
2. *generic operations*, for example generic arithmetic, and
3. *checking operations specified at the source level*, for example the function *atom*.

Ignoring efforts to eliminate type and tag checking at compile-time using type deduction, the cost of tag checking can be reduced in two ways. First, by eliminating the need for tag extraction, thus reducing the cost of tag checking operations in all of the above categories. Second, by eliminating some tag checking operations completely in some of the categories. Because tag checking is a very simple operation, there is no room for software optimization, and both approaches require hardware changes. Both strategies are discussed in this section.

6.1. Eliminating tag extraction

We saw in Section 3.4 that tag extraction followed by a single eq/neq comparison is the most common form of type checking. The tag extraction operation can be avoided with a special conditional eq/neq branch that only tests the part of the word that contains the tag. This would eliminate almost 4% of the instructions, if no run-time checking is done, and about 10% of the instructions with full run-time checking. Some architectures, for example the VAX-11, have instructions that compare bit-fields directly, and that allow tag checking without explicit tag extraction. However, these instructions must be faster than a sequence of simpler instructions if they are to yield a performance improvement.

The special conditional branches can be implemented in a number of ways. First, it can be hardwired into the processor what bits will be used in the comparison. This is not very flexible, because the architecture, and not the software, determines where the tag bits should be placed in the word, but such an implementation is acceptable for LISP machines [14, 6, 21]. Second, the special conditional branch can take a mask as a third argument. This solution is more flexible, but it is expensive to implement because it introduces an instruction with 3 sources, complicating the data path and shortening the branch offset. Third, the bits can be specified by a mask that is set under program control. This can either be implemented by having a special mask register, or by using a specific general purpose register, in which case we would still need 3 reads from the register file. Note that an eq/neq comparison on part of a word, specified by a mask, is easy to implement and is also very fast.

6.2. Hardware tag checking

The cost of tag checking can be substantially reduced by providing special hardware that does some tag checking operations in parallel with other operations. It is not really possible to eliminate tag checking operations that are specified in the source code (category three of the beginning of this section), but LISP machines such as the Symbolics 3600 [14], TI Explorer [6] and SPUR [21] provide hardware support for tag checking operations that result from error checking on primitive operations, and from generic operations (categories one and two).

BNSDOCID <XP 1709A I >

6.2.1. Error checking

Software tag checking used for error testing on primitive LISP operations can be eliminated by having special memory instructions that check the tag of the address during address calculation, and that trap if the tag does not have the expected value; the expected tag could be specified in a register, as an immediate, or in the opcode. The hardware test is limited to a simple tag check; for operations such as list accessing, this is sufficient, but for vector operations, range checking would still have to be done in software.

Hardware for parallel error checking is very simple if the tag location and the tag values, can be built into the hardware. For example, if the expected data type is encoded in the opcode, a PLA with the opcode and the operand tag as inputs, and a single output indicating success or failure, is all that is needed. The hardware becomes a lot more complicated if the tag implementation has to be left to the software.

For MIPS-X specifically, two difficulties would arise when adding parallel tag checking. First, the path to load/store data from/to the cache is critical; adding gates to that path to implement parallel checking and trapping, would slow down the processor. Second, the MIPS-X design tries to detect the arrival of an exception (e.g. traps and interrupts) as early as possible, to keep the exception handling out of the critical path of the processor. Adding an instruction that could explicitly trap, might reduce the time that is available to handle exceptions, thus slowing down the processor.

Run-time error checking on list operations accounts for between 0% and 12% of the execution time in our programs. Adding hardware to do this test in parallel with the address calculation would eliminate this cost, plus an extra 0%-4% because no tag removal would be required for memory accesses to lists. Extending the hardware to allow parallel type checking for other data types (vectors and structures), could give a speedup similar to the speedup for lists, depending on what data types are used in the program.

Note that the MIPS-X architecture naturally allows some overlap between an operation and its corresponding tag checking operation. With a *squashed delayed branch*, two instructions are executed while the branch condition is calculated and while the next instruction is fetched, and the effect of both instructions is cancelled if the the branch does not go [5]. An operation and its tag check will happen concurrently, if the branch condition is 'tag equal to expected tag' and if the operation is moved in a delayed slot of the branch.

6.2.2. Generic operations

Generic operations are operations that can handle data of different types. Examples are generic arithmetic and the function *equal*. Generic operations can be implemented in software by testing sequentially for the different possible data types, by dispatching on the type of the operand(s) (basically a case statement), or by a combination of the two, as described in see Section 2.2 for integer-biased generic arithmetic. It is possible to provides hardware support for generic operations at various levels. A first possibility is to test the type of the operands, while executing the operation, assuming that the operands are of the most common type. If the test fails, a trap is generated, and the operation is aborted; less common data types can then be handled in software. The implementation of arithmetic operations on SPUR [21] follows this strategy. This approach is very fast for the most common data type, but the treatment of other data types can be slow, depending on how fast traps are handled, and on how often the 'less common' case occurs. A floating point program that uses integer-biased generic arithmetic could well be slower than an integer-biased software implementation, because of the trap overhead. Trap handling can be simplified, at the expense of extra hardware, by the use of shadow registers that cache the operands [22].

By also providing hardware support for dispatching on the type of operands, it is possible to further speed up generic operations [14, 6]. The VLSI chip used in the TI Explorer II [2], for example, has a special on-chip memory for dispatch tables that can be used by the micro-code. Generic arithmetic operations are implemented by testing for the most common integer case while starting the integer operation, and if the test fails, the micro-code dispatches on the type of the operands. This approach is similar to the software implementation of generic arithmetic on general-purpose processors, but it will be faster, if the extra hardware is free, because it allows more parallelism.

The hardware described in this section would reduce the cost of generic arithmetic to 1.3%, down from 2%; all operands are integers, so a type dispatch is never needed. If a type dispatch is needed, the performance of the different tag implementations will vary strongly. When a type dispatch is needed for every arithmetic operation, that is, the inline test always fails, then with the MIPS-X software tag implementation, the overhead of the type dispatch would increase the average execution time by 2.7%. We expect that this number will be lower on a processor like the Symbolics, but it will be higher if less common data types cause a trap, as in SPUR.

Compile-time analysis can be used to reduce the cost of using the wrong bias. If compile-time analysis indicates that the operands are probably not integers, the compiler can generate code that invokes the general dispatch routine, or a (software) routine with a different bias. As the compiler is more and more successful at deriving data types, the 'less common' case will become more and more an exception.

BNSDOCID: <XP_ 1709A I_>

8283 Operating Systems Review
21 (1987) Oct., No.4, Washington,DC,USA

7. Summary: what can hardware buy?

Table 2 shows what fraction of the cycles (after pipeline scheduling) would be eliminated in the ten PSL programs, with the various software and hardware tag implementations discussed in this paper. The programs were executed as described in Section 3, and the speedup is relative to the execution time with the straightforward tag implementation of Section 2.1. The two columns give results for programs with, and without run-time checking, and each column gives the speedup relative to the execution time for that column. The exact speedup will depend on the quality of the compiler, and on whether the compiler has been tuned for speed or for safety. Remember that adding full run-time checking in software slows down our programs by 25% on average.

The first three rows summarize the results for the tag implementations of Sections 5 and 6.1:

- row one corresponds to a software tag implementation in which the tag is kept in the bottom two or three bits of the word so no explicit tag removal is necessary before accessing memory; load and store instructions that ignore the tag bits in the address give the same speedup.
- row two gives the speedup if the processor has a special conditional branch that checks the tag without extracting it.
- row three gives the result if the implementations of rows one and two are combined.

These three implementations require either no hardware changes, or very simple hardware changes.

The fourth row shows the reduction in cycles if we had special hardware that traps if an arithmetic operation has non-integer operands, or if overflow occurs. The speedup is small because our programs are not computation intensive; note that the tag implementation of Section 4.2 would further reduce the gain of hardware support for generic arithmetic over a software implementation, but that tag implementation is not compatible with the implementation of row 1. The fifth row gives the speedup if tag checking on list operations is done with extra hardware in parallel with the address calculation (Section 6.2). In the sixth row, we assumed that parallel tag checking is not only possible for lists, but also for vectors and structures. These hardware additions are more substantial because they influence the control of MIPS-X.

The seventh row corresponds to a processor that has:

1. loads and stores that ignore the tag (row 1),
2. special instructions to check the tag without extracting it (row 2),
3. hardware for generic arithmetic (row 4),
4. loads and stores that do parallel error checking for all data types (row 6).

This is to the maximum amount of hardware support that can be added to MIPS-X without requiring a total reorganization of the processor. It would eliminate between 9% and 22% of the cycles. This savings should be compared with the software implementation of row one (6%-5%), and with row three (9%-14%), which only requires very limited hardware changes.

			no run-time checking	run-time checking
-1-	avoid tag masking (software)		5.7%	4.6%
-2-	avoid tag extraction		3.6%	9.3%
-3-	avoid masking and extraction		9.3%	13.9%
-4-	support generic arithmetic		0%	0.7%
-5-	avoid tag checking on list ops	check	0%	12.1%
		mask	0%	4.2%
		total	0%	16.3%
-6-	avoid error tag checking (lists+vectors)	check	0%	13.6%
		mask	0%	4.6%
		total	0%	18.2%
-7-	avoid masking		5.7%	4.6%
	avoid extraction		3.6%	2.9%
	avoid all error tag checking		0.0%	14.6%
		total	9.3%	22.1%

Table 2: Speedup in percent for different degrees of hardware support

NSDOCID <XP 1708A I >

More complex hardware support, such as microcode support for type dispatching, is possible. However, it would require a total reorganization of the MIPS-X architecture, and our results indicate that the payoff would be much less than the negative impact of the added hardware on cycle time and other performance measures.

SPUR [21] provides hardware support for tag handling corresponding to row seven, except that SPUR does not allow parallel checking on memory accesses other than list accesses (row 5 instead of row 6). As a result, the SPUR tag hardware would eliminate between 9% and 21% of the cycles in our programs, although the gain drops to between 4% and 16% of the cycles, if the software tag implementation of row one is used on MIPS-X.

8. Conclusion

In this paper we first looked at the cost of the various tag handling operations, if a straightforward tag implementation is used. We found that tag checking, which includes the cost of extracting the tag, is the most expensive operation, certainly if full checking at run-time is required (11%-24% of execution time). Tag removal, which is done before using the data, uses about 8% of the execution time, and tag insertion uses 1.5%.

Then, we looked at how the cost of tag handling can be reduced with different software and hardware tag implementations. The cost of tag removal can be reduced in software by putting the tag in the bottom 2 or 3 bits of the data word; this results in a speedup of about 5%. Speeding up tag checking requires special hardware. One possibility is to eliminate the need for tag extraction before a tag check. This is easy to implement, and it gives a speedup of 4% to 9%, depending on how many tag checking operations can be eliminated by the compiler. The two optimizations combined eliminate between 9% and 14% of the cycles. Another possibility is to have special hardware that does tag checking in parallel with memory access operations. This, together with the previous features, gives a speedup of between 9% and 22%, but it requires more complicated hardware, and the tag implementation has to be built into the architecture.

Appendix

The following set of 10 LISP programs were used to collect data:

- *inter*: a simple interpreter for a subset of LISP is used to calculate the Fibbonachi number 10, and to sort a list of numbers; adapted from "Lisp in Lisp" [25].
- *deduce*: a deductive information retriever for a database organized as a discrimination tree; adapted from [4].
- *dedgc*: the same program as *deduce*, but a copying garbage collector is invoked. The program spends about 50% of its time in the garbage collector.
- *rat*: a rational function evaluator that comes with the PSL system.
- *comp*: the first pass of the front-end of the PSL compiler.
- *opt*: the optimizer that was added to the compiler. It uses lists, and vectors.
- *frl*: a simple inventory system using the *frame representation language*.
- *boyer*: the boyer benchmark; a rewrite-rule-based simplifier combined with a dumb tautology-checker; benchmark published by Gabriel [7].
- *brow*: a short version of the browse benchmark; creates and browses through an AI-like database of units; benchmark published by Gabriel [7].
- *trav*: a short version of the traverse benchmark; creates and traverses a tree structure; uses structures which are implemented as vectors; benchmark published by Gabriel [7].

Table 3 gives the number of procedures, the number of lines of source code (without comments), and the number of MIPS-X machine instructions after compilation, for each program. Each program includes besides the user program, the LISP system modules, or parts of modules, that are used by the program.

	number of procedures	lines source code	words object code
inter	64	710	1533
deduce	100	900	3419
dedgc	116	1100	4112
rat	148	1900	6315
comp	220	2400	9466
opt	226	3500	11121
frl	198	2500	11802
boyer	84	1200	1793
brow	91	1000	2296
trav	78	810	1673

Table 3: Information on the 10 test programs

Acknowledgments

We thank the members of the MIPS-X group for their help and suggestions. Mark Horowitz provided helpful information about the MIPS-X implementation. The PSL system was developed at the University of Utah.

BNSDOCID <XP 1708A I >

References

1. Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., and Weinreb, D. LISP Machine Progress Report. Memo No 444, MIT Artificial Intelligence Laboratory, August, 1977.

2. Bosshart, P., Hewes, C., Chang, M., and Chau, K. A 553K-Transistor LISP Processor Chip. Digest 1987 International Solid-State Circuits Conference, IEEE, New York, February, 1987, pp. 202-203.

3. Brooks, R., Posner, D., McDonald, J., White, J., Benson, E., and Gabriel, R. Design of An Optimizing, Dynamically Retargetable Compiler for Common Lisp. Proceedings of the 1986 Conference on LISP and Functional Programming, ACM, Boston, August, 1986, pp. 67-85.

4. Charniak, E., Riesbeck, C. K., and McDermott, D. V.. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

5. Chow, P., and Horowitz, M. Architectural Tradeoffs in the Design of MIPS-X. Proceedings of the 14th Annual International Symposium on Computer Architecture, ACM, June, 1987, pp..

6. Corley, C.J. and Statz, J.A. "LISP workstation brings AI power to a user's desk". *Computer Design 24*, 1 (January 1985).

7. Gabriel, R.P.. *Computer Systems Series*. Volume : *Performance and evaluation of LISP systems*. The MIT Press, 1985.

8. Griss, M.L. and Hearn, A.C. "A Portable LISP Compiler". *Software - Practice and Experience 11*, 6 (June 1981), 541-605.

9. Griss, M.L., Benson, E., and Maguire, G.Q. PSL: A Portable LISP System. Proceedings of the 1982 Symposium on LISP and Functional Programming, Pittsburgh, August, 1982, pp. 88-97.

10. Horowitz, M., Hennessy, J., Chow, P., Gulak, P., Acken, J., Agarwal, A., Chu, C.Y., McFarling, S., Przybylski, S., Richardson, S., Salz, A., Simoni, R., Stark, D., Steenkiste, P., Tjiang, S., and Wing, M. A 32b Microprocessor with On-Chip 2K Byte Instruction Cache. Digest 1987 International Solid-State Circuits Conference, IEEE, New York, February, 1987, pp. 30-31.

11. Kranz, D., Kelsey, R., Rees, R., Hudak, P., Philbin, J, and Adams, N. ORBIT: An Optimizing Compiler for Scheme. Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, ACM, Palo Alto, June, 1986, pp. 219-233.

12. Milner, R. "A Theory of Type Polymorphism in Programming". *Journal of Computer and System Science 17*, 3 (December 1978), 348-375.

13. Moon, D.A. *Maclisp Reference Manual*. MIT, Laboratory of Computer Science, 1983.

14. Moon, D.A. Architecture of the Symbolics 3600. Proceedings of the 12th Annual International Symposium on Computer Architecture, ACM, Boston, June, 1985, pp. 76-83. Also in SIGARCH Newsletter 13(3).

15. Rees, J., and Adams, N. T: A Dialect of Lisp or, or LAMBDA: The Ultimate Software Tool. Proceedings of the 1982 Symposium on LISP and Functional Programming, Pittsburgh, August, 1982, pp. 114-122.

16. Steele, G.L. Jr. Fast Arithmetic in MacLISP. Proceedings of the 1977 MACSYMA Users' Conference, July, 1977.

17. Steele, G. L. Jr.. *Common Lisp - The Language*. Digital Press, 1984.

18. Steenkiste, P., and Hennessy, J. LISP on a Reduced-Instruction-Set-Processor. Proceedings of the 1986 Conference on LISP and Functional Programming, ACM, Boston, August, 1986, pp. 192-201.

19. Steenkiste, P. *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization*. Ph.D. Th., Stanford University, March 1987.

20. Stritter, E., and Gunter, T. "A Microprocessor Architecture for a Changing World: The Motorola 68000". *IEEE Computer 12*, 2 (February 1979), 43-52.

21. Taylor, G.S., Hillfinger, P.N. Larus, J., et al. Evaluation of the SPUR Lisp Architecture. Proceedings of the 13th Annual International Symposium on Computer Architecture, ACM, Tokyo, June, 1986, pp. 444-452.

22. Ungar, D. *The Design and Evaluation of A High Performance Smalltalk System*. Ph.D. Th., UC Berkeley, March 1986. Technical Report UCB/CSD 86/287.

23. White, J. "LISP/370: A Short Technical Description of the Implementation". *SIGSAM 12*, 4 (November 1978), 23-27.

24. White, J. Reconfigurable, Retargetable Bignums. Proceedings of the 1986 Conference on LISP and Functional Programming, ACM, Boston, August, 1986, pp. 174-191.

25. Winston, P. and Horn, B.. *Lisp*. Addison-Wesley Publishing Company, 1981.

INSDOCID <XP 1708A I >

XP-002161891

P.D. 23-06-1987

P. ....

(8) G06F12/02D2

E p. 230-236.

CONCURRENT GARBAGE COLLECTION WITH ASSOCIATIVE TAG

Heonshik Shin*, Miroslaw Malek+, Sukho Lee*

Department of Computer Engineering*
Seoul National University
Seoul 151, Korea

Department of Electrical and Computer Engineering+
University of Texas at Austin
Austin, Texas 78712, U.S.A.

Lisp

ABSTRACT

An algorithm for efficient concurrent garbage collection is presented and analyzed. The algorithm is based on the concept of tagged memory where each element is combined with an associative tag to denote its property. The associative tag is used to indicate whether the corresponding element is free for allocation, marked as an active node, and queued for marking. The garbage collection (GC) process marks a queued node and queues the unmarked successors, whereas the list manipulation (LM) process queues the candidate nodes for marking using the tag and the primitive operations on it. The correctness of the algorithm is proved. It is claimed that according to the proposed algorithm (1) no free list is necessary for memory allocation, (2) fast marking is achieved without a stack or the critical section, and (3) the LM and GC processes would experience fewer memory access conflicts. List compacting is not considered here.

I. INTRODUCTION

Lists are the fundamental data structure in many areas of computation, especially in artificial intelligence programs. Good facilities for manipulating lists are essential to the machines for such computation. From the list management point of view, list processing can be functionally dichotomized into list manipulation and garbage collection. A list manipulation process (LM process) is responsible for creating and maintaining lists. Since the list processing usually relies on the dynamic memory allocation, list nodes should be available for allocation at run-time. Meanwhile, the used nodes are deallocated implicitly without the knowledge of the LM process or operating system. These deallocated but unidentified list nodes (called garbage nodes) need to be reclaimed for recycling under the constraint of limited storage space. Thus a garbage collection process (GC process) is responsible for identifying and collecting these nodes.

In the classical reclamation scheme [13], the GC process is activated only after the depletion of free available nodes. It starts with the marking of all accessible nodes. The nodes left unmarked at the end of the marking phase are regarded as garbage nodes. These nodes are then appended to the free list which is a linear list of free available nodes. Upon completion of reclamation, the LM process is resumed. Although implementation of the classical scheme is straighforward, the major disadvantages include degraded performance and unpredictable response time. The latter problem in particular can be resolved by using interleaved collection method in which the LM and GC processes time-share the processor, as described by Bobrow [3] and Knuth [8, p.422, p.594]. This method called incremental garbage collection guarantees continuous run of a user program. However, the incremental collection scheme causes greater delay in list processing than the classical scheme [21].

Concurrent garbage collection resolves these two problems: inefficiency and unpredictability of response time. The LM process and the GC process are run on separate processors while sharing a common list memory. Various concurrent algorithms have been proposed based on the mark-and-collect method of McCarthy [13]. In general, there are two basic approaches to this problem: coloring method using two tag bits [4, 5, 11], and the stacking method [7, 10, 14, 20, 21]. The former requires no critical sections, thus offering conflict-free memory access to both processes. However, the time necessary for marking is O(N) to O(NA) where N is the total number of nodes in the list memory and A is the number of active nodes accessible to the LM process. Recently, similar but less efficient algorithm is proposed, requiring a single tag bit [2]. The latter stacking technique offers efficient marking requiring O(A) time, but it demands extra memory space for the stack which must be declared as the critical section.

CH2433-1/87/0000/0230S01.00 © 1987 IEEE

ZIE VOOR TITEL BOEK, de 2e PAGINA.

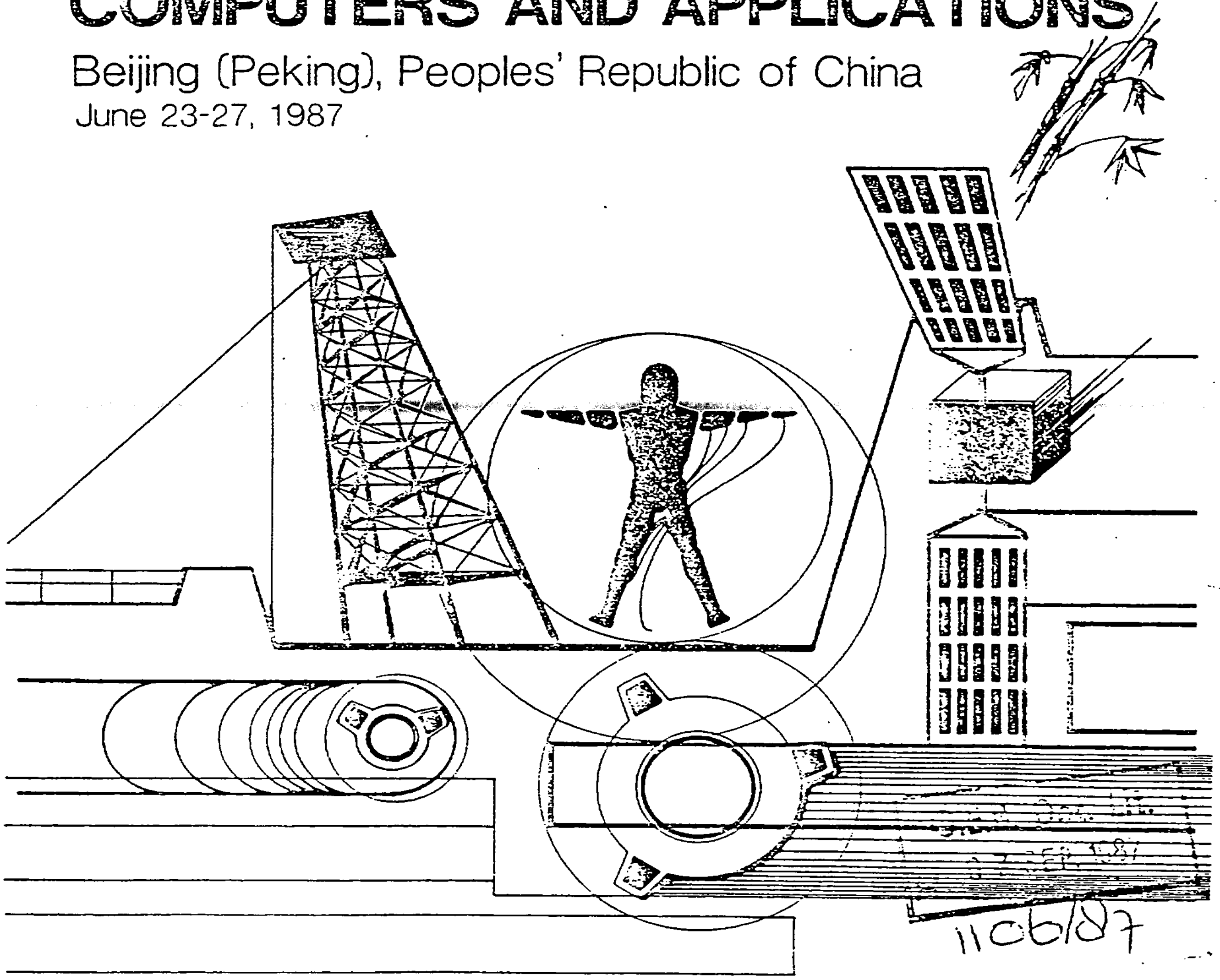
BNSDOCID <XP 2161891A I >

The Second International Conference on

COMPUTERS AND APPLICATIONS

Beijing (Peking), Peoples' Republic of China

June 23-27, 1987



Best Available Copy

Computer Society Order Number 780
Library of Congress Number 87-60478
IEEE Catalog Number 87CH2433-1
ISBN 0-8186-0780-7
SAN 264-620X

Co-Sponsored by Chinese Computer Federation

The Computer Society of the IEEE

In cooperation With
The National Natural Science Foundation of China

THE COMPUTER SOCIETY OF THE IEEE

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC

COMPUTER SOCIETY PRESS

NSDOCID <XP 2181891A I >

Modified versions [9, 15] have been proposed to achieve efficient marking in O(A) time; however, they have failed to eliminate the need for significant extra memory space for marking. This is considered a serious drawback because shortage in memory is the very reason that the garbage collection is needed. Moreover, the possible saving of space accrued from faster marking (as observed in [21]) may be offset by the requirement of extra space for marking.

In this paper, an efficient algorithm for concurrent garbage collection is developed based on the mark-and-collect method. Using extensively a two-bit associative tag which may be addressed by its contents, they combine the coloring method and the stacking method. Also, in general, the algorithm accomplishes performance improvements for the memory allocation; in particular, efficient marking in O(A) time with no critical section or additional space required, and economical management of free available nodes. List compacting is not considered here because a garbage collection scheme with the marking and collection phases is perfectly satisfactory.

In what follows, definitions and assumptions are given in the next section. Sections 3 and 4 describe the respective procedures necessary for the LM process and the GC process. Section 5 is provided with the proof of correctness of the concurrent algorithm. Section 6 shows the effect of the algorithm on system performance. Conclusions are drawn in Section 7.

II. DEFINITIONS AND ASSUMPTIONS

2.1 List Representation

A list structure is in essence nothing but a collection of list nodes in which each node usually contains two pointers to other nodes. There are two types of list nodes: pointer node and data node. A pointer node is a number of contiguous words which consists of two pointers, LLINK and RLINK. A data node contains a value; so, it is always a terminal node. If there are no list nodes to point to, the pointer (LLINK or RLINK) has the value NIL. Note that the header node of a list called the root node is accessible to every node in that list.

The list nodes are linearly labeled in the list memory. Each node has an associative tag. From the memory allocation point of view a node falls into one of the following classes:

(a) Active nodes: the nodes which are accessible from at least one of the root nodes.

(b) Free nodes : the nodes which are available for allocation.

(c) Garbage nodes : the nodes which are inaccessible from any roots and are not available for allocation.

The major goal of garbage collection is to determine the accessibility of nodes and distinguish between active nodes and garbage nodes. This requires the traversal of the list graph which is discussed in the next section. Dynamic memory allocation in list processing changes the node type at run-time as follows:

```
      ------------- allo-     ---------------
--->| free node |-------->| active node |--
|     ------------- cation   ---------------  |
|                                             |
|                                             |
|                                             |
|  garbage  ----------------  deallo-         |
-----------| garbage node |<--------------
 collection ----------------  cation
```

2.2 State of List Nodes

At any instant of time, a node is in a certain state. The state of a node tells which type it is, and changes as list manipulation and garbage collection progress in time. Its state may be defined by the state vector which consists of the following state variables.

(a) Fvar: indicates whether a node is free (available) for allocation. If available, then Fvar=1; else Fvar=0.

(b) Mvar: indicates whether a node is marked as the active node. If marked, then Mvar=1; else Mvar=0.

(c) Qvar: indicates whether a node is queued for marking in the foreseeable future. If queued, then Qvar=1; else Qvar=0.

A triplet (Fvar, Mvar, Qvar) represents the state vector of a node. Four states are defined in terms of the state vector as follows:

State Name	State Vector	Description
Fstate	(1,0,0)	free node
Mstate	(0,1,0)	active node, marked
Qstate	(0,0,1)	active node, queued for marking
Gstate	(0,0,0)	garbage node

2.3 Associative Tag

An associative tag is the tag in the memory whose address can be identified based on its content. A list node is associated with an associative tag to denote its state. We consider three parameters in relation to the associative tag: address, state(content), and a

Best Available Copy

BNSDOCID <XP 2161891A I >

Boolean variable "empty." The following operations are defined on the tag. Separated by a vertical bar, the first set of arguments of each operaton represents input parameters; the second set represents the output parameters.

(a) Read(addr|state): Read and return the content state at location addr.

(b) Write(addr,state|) : Replace the content with state at location addr.

(c) Find(state|addr,empty): Search for any one of the nodes which are in state, and return its address addr with empty set to false. If the search fails, return empty set to true.

The associative tag, especially the Find operation, can be effectively implemented using the Boolean content addresable memory[17].

2.4 Assumptions

(1) The minimum time require for the creation of a list element is greater than the maximum interval between two consecutive instances of marking a node.

(2) The primitive operations are atomic; so execution of each operation is uninterruptable.

(3) A set of root nodes for all lists is provided for the GC process; so all the active nodes are accessible to the root nodes.

(4) The content of data nodes is NIL.

III. LIST MANIPULATION PROCESS

3.1 Allocation of Free Nodes

The LM process carries out two important functions in relation to concurrent garbage collection: allocation of free nodes and cooperation with the GC process in identifying garbage nodes. A linear list called free list has been used conventionally for the allocation of free nodes so that all free nodes may be traced. Alternatively, a bit-map or bit-table may be used where a bit is reserved for each list node to indicate whether it is free or active [6]. Our approach makes use of the associative tag so that free nodes have their tag set to Fstate.

When a free node is requested for the creation of a list element, the memory is searched for a node which is in Fstate. Once found, its state is changed to Qstate to ensure that the node will be marked as an active node by the GC process. So when a new list element is created, the address of a free node is provided according to the procedure shown in Fig. 1a. It is assumed that when the "empty" variable is false indefinitely, appropriate actions are taken by the operating system to avoid the infinite loop.

3.2 Redirecting of Pointers

Identification of garbage nodes is achieved by marking all active nodes. In order to cooperate with the GC process for correct identification of garbage nodes, the LM process should ensure that when a pointer is modified the node pointed to by the new pointer may be marked as an active node. The redirecting of pointers is defined as having an existing pointer directed to another active node.

To do so, whenever a pointer is redirected to point to an active node n_a , n_a must be queued for marking if it has not been marked. A procedure for the pointer alteration must include the provisions shown in Fig. 1b.

```
PROCEDURE FreeNodeAddress(addr);
 BEGIN
  REPEAT
   Find(Fstate|addr,empty)
  UNTIL empty = False;
  Write(addr,Qstate|)
 END
```

(a) Procedure P1: Allocation of free nodes

```
PROCEDURE QueueIt(newpointer);
  (* "newpointer" is a redirected pointer.*)
BEGIN
 Read(newpointer|state);
 IF state = Gstate THEN
   Write(newpointer,Qstate|)
END
```

(b) Procedure P2: An auxiliary to pointer alteration

Figure 1: Procedures P1 and P2 for the LM process

IV. GARBAGE COLLECTION PROCESS

4.1 Pseude-Random Search

Pseudo-random search (PRS) forms the foundation of marking algorithm for concurrent garbage collection. Using an undirected graph whose vertices have three possible states, namely, Gstate, Mstate and Qstate, we now summarize the PRS algorithm proposed in [17].

The search begins with setting the state of all vertices to Gstate. Next the start vertex is put in Qstate. Then as a vertex in Qstate is selected and visited, its adjacent vertices in Gstate are all placed in Qstate. This cycle repeats until no more vertex in Qstate is left. The algorithm is given below.

BNSDOCID <XP 2161891A I >

```
PROCEDURE PseudoRandomSearch(startvertex)
  BEGIN
    FOR each vertex DO set its state to Gstate;
    Set the state of startvertex to Qstate;
    WHILE there is a vertex v in Qstate DO
      BEGIN
        Set the state of v to Mstate;
        FOR each vertex w adjacent to v DO
          Set state of w to Qstate if in Gstate;
      END
  END
```

4.2 Garbage Collection Algorithm

The GC process consists of two phases: marking and transformation. In the marking phase, all active nodes are marked so unmarked nodes are identified as garbage nodes. The marking process follows the PRS: given the root node of each list structure, the GC process traverses the structure using the PRS technique. In the transformation phase, garbage nodes are transformed into free nodes. So all the nodes in the list memory that have not been marked in the marking phase are incorporated into the free available space. The garbage collection procedure P3 is shown in Fig. 2.

The GC process begins by initializing all list nodes except free nodes to Gstate. In the marking phase, each rooted list structure is thoroughly traversed as observed in the PRS. The active nodes shared by two or more lists are marked once since the nodes in Qstate or Mstate are not allowed to be placed in Qstate again. The marking is irreversible in the sense that once a node is marked it can not be "unmarked" until the next garbage collection even though the node thereafter becomes garbage.

Next in the transformation phase, all the nodes in Gstate are assigned Fstate. This indicates that a node is not free if it has been marked (Mstate) or queued for marking (Qstate). The latter case is to prevent the node just allocated after the marking phase from being transformed into free node. In summary, all the garbage nodes present at the beginning of a garbage collection are transformed into free node after the end of it.

V. CORRECTNESS OF THE ALGORITHM

Correct garbage collection algorithms preserve the integrity of list structures throughout the GC process. The LM process and the GC process should cooperate in such a way that

- The list structure should not be modified as a result of garbage collection.
- The garbage collection terminates properly so that the garbage nodes are transformed into free node after predictable amount of time.

```
PROCEDURE GarbageCollection(root);
 BEGIN      (*Initialization*)
  FOR i:=1 to N DO
    BEGIN
      Read(i|state);
      IF state <> Fstate THEN
        Write(i,Gstate|)
    END;
                              (*MARKING PHASE*)
  FOR i:=1 to M DO (*M = number of roots*)
    BEGIN
     node := root[i];
     REPEAT
      Write(node,Mstate|);
      FOR each link DO
        BEGIN
          succesor := SUCC(node,link);
          IF succesor <> NIL THEN
            BEGIN
              Read(successor|state);
              IF state = Gstate THEN
                Write(successor,Qstate|)
            END
        END;
      Find(Qstate|node,empty)
     UNTIL empty = True
   END;
                    (* TRANSFORMATION PHASE*)
   FOR i :=1 to N DO
      BEGIN
        Read(i|state);
        IF state = Gstate THEN
          Write(i,Fstate|)
      END
   END
```

(*The function SUCC returns the address of the successor node specified by the parameters.*)

Figure 2: Procedure P3 for the GC process

- No active nodes should be mistaken for garbage nodes.

In the following, these requirements are investigated according to the procedures P1, P2 and P3.

5.1 Proper Manipulation of List Structures

The list structures are manipulated by the LM process as a garbage collection proceeds. So the GC process should not interfere with the LM process in manipulating the lists.

Theorem 1: No list structures are modified as a result of the garbage collection.

Proof: The list structure is mutated when a node or a pointer is added or removed. The parallel procedures P1 through P3 do not involve any operations on the pointer or value in a node except for the function SUCC in P3. This function, however, only fetches a pointer in a given node, and does not alter its content. Hence no nodes or pointers are modified by the algorithm.□

Best Available Copy

BNSDOCID <XP 2161891A I >

5.2 Proper Termination

Only the marking phase is considered for the proper termination of the GC process because the time required for the transformation is O(N) according to procedure P3. Let

T_a = the minimum interval between the two consecutive instances of allocating a new node,
T_m = the maximum interval between two consecutive instances of marking a node,
R = $T_m/T_a < 1$ by the assumption given in Section 2,
N_0 = the number of active nodes at the beginning of the marking phase.

Lemma 2: The time required for the marking phase is less than or equal to $T_m N_0 (1-R)^n/(1-R)$ if the active nodes are marked once, where $n = \lfloor 1-\log N_0/\log R \rfloor$.

Proof: All active nodes that are marked by the GC process have been either active or newly allocated since the beginning of the marking phase. The total number, N_t, of active nodes can be calculated for the worst case where new active nodes are always created and no existing active nodes are deallocated while the marking proceeds. In other words, every time N_i active nodes are marked t_i/T_a nodes become active anew. t_i stands for the maximum time required to mark N_i nodes. Note that $N_i > N_{i+1}$ because $T_a > T_m$. The following shows how the marking progresses in time.

$$
\begin{array}{lllll}
N_i: & N_1 = N_0 & N_2 = t_1/T_a & \ldots & N_n = 1 \\
t_i: & t_1 = N_1 T_m & t_2 = N_2 T_m & \ldots & t_n = T_m
\end{array}
$$

where in general $N_k = t_{k-1}/T_a$ and $t_k = N_k T_m$. Hence the total number of active nodes for marking is the sum of a geometric series.

$$
\begin{aligned}
N_t &= N_1 + N_2 + \ldots + N_n \\
&= N_0 + N_0 T_m/T_a + \ldots + N_0 (T_m/T_a)^{n-1} \\
&= N_0 (1-R)^n/(1-R)
\end{aligned}
$$

where the integer n must be the largest integer that satisfies the relationship $N_0 R^{n-1} \geq 1$. Thus the total time necessary for the marking phase in the worst case is $T_m N_t = T_m N_0 (1-R)^n/(1-R)$ where $n = \lfloor 1-\log N_0/\log R \rfloor$. □

Theorem 3: The garbage collection terminates properly.

Proof: The time required for the initialization and the transformation in the procedure P3 is clearly O(N). According to Lemma 2 the garbage collection terminates within the finite period of time if active nodes are marked once. Nodes are marked if they are in Qstate except for the root nodes. The procedures P1 through P3 indicate that Qstate is assigned only when the nodes have never been marked or queued. As a result each node is queued for marking once, thus marked once. □

5.3 No Mistaken Active Nodes

If there were any active nodes mistaken for garbage, these active nodes must have not been marked during the marking phase. In order to prove that all the active nodes are marked, it is necessary to show that (1) initially all the active nodes except the root nodes are queued for marking, and that (2) once queued, they are all marked. Note that all the root nodes are marked according to the procedure P3.

These tasks can be accomplished by the proper cooperation between the LM process and the GC process. The cooperation centers around the operations on the associative tag. The correctness of their cooperation can be proved by showing that the tag is properly manipulated. State transition occurs when a write operation is performed on the tag of individual nodes. For correct identification of garbage nodes, the state of each node must undergo the transition intended by a given process only. If there are any possibilities of undesirable state transition, the correctness of the algorithm is not guaranteed. The state transitions in the algorithm can be summarized as follows:

```
P1:  Fstate --> Qstate
P2:  Gstate --> Qstate
P3:  Qstate --> Gstate ----> Mstate
             |           |
     Mstate--|           |--> Qstate --> Mstate
                         |
                         |--> Fstate
```

Lemma 4: One process does not interfere with the other process with respect to the state transition.

Proof: We shall consider the LM process vs the GC process, that is, P1 and P2 vs P3. By inspection, the state transitions in P1 and P3 are found mutually exclusive. The transition diagrams of P2 and P3, however, show possible conflicts: as P2 tries to change Gstate to Qstate, P3 may attempt to change Gstate to Mstate or Fstate. The transition of Gstate -> Mstate in P3 occurs only to root nodes regardless of P2. On the other hand, the transition of

Best Available Copy

NSDOCID <XP 2161991A I >

Qstate -> Fstate in P3 occurs in the transformation phase, when all the active nodes have been either marked or queued. Because no state transition is allowed for such nodes according to P2, there are no possibilities of interference. Consequently, the integrity of the state transitions in the two processes are preserved.□

Theorem 5: All the active nodes are queued for marking correctly.

Proof: When every node is marked all of its successors are queued according to the procedure P2. Every active node that is created after the beginning of the marking phase is also queued according to P1. The only case that an active node is not queued is when its sole predecessor (still unmarked) is replaced by another predecessor (already marked). However, this problem is resolved in P2 where a new successor resulting from the redirecting of a pointer is queued for marking explicitly. According to Lemma 4, this is performed correctly.□

Theorem 6: While in the marking phase, all active nodes in Qstate are marked.

Proof: During the marking and transformation phases the transition diagram of P3 shows that the only possible state transitions are from Qstate to Mstate. These transitions occur correctly according to Lemma 4. Hence, all the queued nodes are marked during the marking phase.□

VI. PERFORMANCE IMPROVEMENTS

The concurrent garbage collection proposed in the preceding sections bears the following major advantages. A comparison of the proposed algorithm and other algorithms is given in Table 1.

A. No free list is necessary.

No free list needs to be maintained for memory allocation. The address of a free node can be obtained using procedure P1. Assuming the operation find(state;addr,empty) locates the first free node in memory with an appropriate priority scheme, free nodes are allocated sequentially from lower (or higher) address to higher (or lower) address. Several gains accrue from it. First, the time required to maintain the structure of a free list is saved. It usually involves the deletion and appending of free nodes, and the subsequent updating of pointers. Second, if the free list is implemented as a simple linear list, its header node must be declared the critical section so that the LM and the GC processes may cooperate properly. This scheme obviates such synchronization measure. Last, less frequent references to the pointer part of the memory by the GC process results in less memory access conflicts as discussed below.

Table 1. A Performance Comparison for Parallel Garbage Collection Algorithms

Marking algorithms	Stacking	Coloring	Assoc. tag
Time	O(A)	O(N)-O(N*A)	O(A)
Extra bits/node	1	2	2
Extra space	stack	none	none
Critical section	stack	none	none
Free list	necessary	necessary	none

B. No stack is necessary for efficient marking.

A stack is required for the efficient marking algorithms with O(A) computation time where A is the number of active nodes. In this case, only the active nodes are marked [1, 19]. The stack requires extra space as large as the memory size in the worst case. For this reason, a great deal of effort has been invested in reducing the stack size or obviating the stack itself at the expense of more computation time [10, 12, 16, 22]. As shown in Section 3, the tagged memory achieves the same efficiency as obtained from marking with a stack. Consequently, the proposed marking algorithm requires little extra cost in terms of time and space provided the associative tag is given.

C. No critical sections are necessary.

In contrast with marking algorithms with a stack [6, 19, 20] and memory allocation with a free list, the proposed algorithm has no critical sections. Accordingly, the two concurrent processes do not require any synchronization schemes. Elimination of overhead in synchronization contributes to the speedup of computation, especially, in the LM process.

D. Memory access conflicts are kept low.

The tagged memory is logically divided into two parts: tag and vaule. They can also be implemented as physically separate regions. The LM process mainly works with pointers and data whereas the GC process utilizes the tag except for the addresses of successor nodes at the time of marking. Hence the possibility of the access conflict is considered substantially low.

VII. CONCLUSION

The proposed algorithms for concurrent garbage collection makes use of a two-bit associative tag. First, in this algorithm, the free list need not be maintained for memory allocation. Second, the marking based on the new graph traversal algorithm can be efficiently

Best Available Copy

BNSDOCID <XP _ 2161891A I >

conducted without the need for a stack and the critical section. Third, the list processor and the garbage collector would experience fewer memory access conflicts because of absence of the critical section and possible separation of the pointer region from the tag region. Last, the implementation is straightforward if the CAM and the dual port memory are provided.

This parallel algorithm is suitable for real-time applications where the predictable, short response time is crucial. It may be extended to the case of the concurrent garbage collection with multiple list processors. In this case, the procedure P1 needs to be modified so the free nodes are allocated properly. It may also be extended to parallel garbage collection where multiple garbage collectors run simultaneously [18].

REFERENCES

1. J.-L. Baer and H. Fries. "On the efficiency of some list marking algorithms," In Information Processing 1977, B. Gilchrist, Ed., Notrh-Holland, Amsterdam, 1977, pp. 751-756.

2. M. Ben-Ari. "Algorithms for on-the-fly garbage collection," ACM Trans. Prog. Lang. Syst. 6, 3(July 1984), 333-344.

3. D. G. Bobrow. "Storage management in Lisp," In Symbol Manipulation Languages and Techniques, D. G. Bobrow, Ed., North-Holland, Amsterdam, 1968, pp. 291-301.

4. E. W. Dijkstra, et al. "On-the-fly garbage collection: An exercise in cooperation," Comm. ACM 21, 11(Nov. 1978), 966-975.

5. D. Gries. "An exercises in proving parallel programs correct," Comm. ACM 20, 12(Dec. 1977), 921-930.

6. M. L. Griss and M. R. Swanson. "MBALM/1700: A microprogrammed LISP machine for the Burroughs B1726," Proc. 10th Microprog. Workshop, 1977, pp. 15-25.

7. T. Hibino. "A practical garbage collection algorithm," Proc. 7th Annual Symp. Comput. Archit., 1980, pp. 113-120.

8. D. E. Knuth. The Art of Computer Programming. Addison-Wesley, Reading, MA, 1973. Vol. 1.

9. H. T. Kung and S. W. Song. "An efficient parallel garbage collection system and its correctness proof," Proc. 18th. Annual Symp. Foundations Compt. Sci., 1977, pp. 120-131.

10. T. Kurokawa. "A new fast and safe marking algorithm". Software - Practice and Experience 11 (1981), 671-682.

11. L. Lamport. "Garbage collection with multiple processes: An exercise in parallelism," Proc. Int'l Conf. Parallel Processing, 1976, pp. 50-54.

12. G. Lindstrom. "Scanning list structures without stacks or tag bits," Inf. Process. Lett. 2, 2(June 1973), 47-51.

13. J. McCarthy, et al. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, MA, 1965.

14. K. G. Muller. On the feasibility of concurrent garbage collection, Ph.D. Th., Technische Hogeschool Delft, March 1976.

15. I. A. Newman and M. C. Woodward. "Alternative approaches to multiprocessor garbage collection," Proc. 1982 Int'l Conf. Parallel Processing, 1982, pp. 205-210.

16. H. Schorr and W. Waite. "An efficient machine-independent proce- dure for garbage collection in various list structures". Comm. ACM 10, 8 (Aug. 1967), 501-506.

17. H. Shin. A Boolean Content Addressable Memory and Its Applications. Ph.D. Th., University of Texas at Austin, May 1985.

18. H. Shin and M. Malek. "Parallel garbage collection with assoicative tag," Proc. 1985 Int'l Conf on Parallel Processing, pp. 369-375.

19. T. A. Standish. Data Structure Techniques. Addison-Wesley, Reading, MA, 1980, Chap. 5.

20. G. L. Steele. "Multiprocessing compactifying garbage collection," Comm. ACM 18, 9(Sep. 1975), 495-508.

21. P. L. Wadler. "Analysis of an algorithm for real time garbage collection," Comm. ACM 19, 9(Sep. 1976), 491-500.

22. B. Wegreit. "A space efficient list structure tracing algorithm," IEEE Trans. Comput. C-21, 9(Sep. 1972), 1009-1010.

NSDOCID <XP 2161891A I >

XP-002161892

P.D. 06-09-1999
P. 476-493 (18)

FACADE: A Typed Intermediate Language Dedicated to Smart Cards

Gilles Grimaud[1], Jean-Louis Lanet[2], and Jean-Jacques Vandewalle[2]

[1] Université de Lille, LIFL/RD2P,
Gilles.Grimaud@lifl.fr, http://www.lifl.fr/~grimaud/
[2] Gemplus Research Lab,
Jean-Louis.Lanet@gemplus.com, and jeanjac@research.gemplus.com

Abstract. The use of smart cards to run software modules on demand has become a major business concern for application issuers. Such downloadable executable content needs to be trusted by the card execution environment in order to ensure that an instruction on a memory area is compliant with the definition of the data stored in this area (*i.e.* its type). Current solutions for smart cards rely on three techniques. For Java Card, either an off-card verifier-converter performs a static verification of type-safety, or a defensive virtual machine performs the verification at runtime. For other types of open smart cards, no type-checking is carried out and the trust is only based on the containment of applications. Static verification is more efficient and flexible than dynamic techniques. Nevertheless, as the Java verifier cannot fit into a card, the trust is dependent on an external third-party. In this way, the card security has been partly turned to the outside. We propose and describe the *FACADE* language for which the type-safety verification can be performed statically on-card.

1 Introduction

In this section the specific domain of smart cards is described. For people not aware of smart cards, we briefly review the technology and the history of smart cards from embedded devices dedicated to specific applications up to open platforms for enabling the downloading of new services all during the card's life.

1.1 The Specific Domain of Smart Cards

Smart cards form a specific domain by three ways we detail hereafter: their internal constitution, their interfaces, and their applications.

A smart card is a piece of plastic, the size of a credit card, in which a single chip microcontroller is embedded. Usually, microcontrollers for cards contain a microprocessor (8-bit ones are the most widespread, but 16-bit and 32-bit processors can now be included in the chip) and different kinds of memories: RAM (for run-time data), ROM (in which the operating system and the basic applications are stored), and EEPROM (in which the persistent data are stored).

BNSDOCID <XP 2161892A I >

Since there are strong size constraints on the chip, the amounts of memory are small. Most smart cards sold today have at most 512 bytes of RAM, 32 KB of ROM, and 16 KB of EEPROM. This chip usually also contains some sensors (like light sensors, heat sensors, voltage sensors, *etc.*), which are used to deactivate the card when it is somehow physically attacked.

In order to be usable, a smart card must be inserted in a card reader/writer, which provides power to the card, as well as a clock signal. Also, any communication between the terminal and the card goes through the card reader/writer in the form of messages exchanged from the terminal to the card (commands), and respectively, from the card to the terminal (responses). All these basic aspects are strongly standardized, since cards are meant to be usable with a wide range of devices. The family of ISO 7816 standards are the references [3]. They standardize many features, from the positions of the contacts on the card to the transport protocol that is used to communicate between the card and the terminal.

A smart card can be viewed as a "data safe", since it stores data in a secure manner and it is used securely during short transactions. Its hardware is the base for its safety. The fact that the chip in a card is embedded with sensors in plastic and glue, and that all components are on the same chip makes a physical attack quite difficult. The software is the second barrier for its safety. The chip programs are usually designed for neither returning nor modifying sensitive information prior to be sure that the operation is authorized. In fact, most card applications use the card either to safely store data, or to process sensitive data. Most smart cards include some support for cryptographic functions, which allows them to secure their transactions with the outside world. More practically, cards are often used either to manage some kind of currency (money or tokens) or to identify a person.

1.2 Smart Cards State-of-the-Art

The specific domain of smart cards is close to the domain of embedded devices. Like embedded devices, smart cards are aimed toward the consumer electronics market, which requires from these systems even more and more convenience and flexibility.

The methods, languages and tools for developing a smart card system share some characteristics with those of the embedded domain. Until recently, smart card codes were written in hand coded native assembler. All programs (drivers, operating system, libraries, applications) were developed in a monolithic piece of code burned in the ROM of the smart card. Therefore, not only traditional card systems are difficult to develop (low-level programming language, very reduced-feature microcontroller, specific code for every microprocessor) but also they cannot support evolution of their applications since all the application code is burned forever with the runtime engine in the ROM.

Moreover, the production of such a "petrified" monolithic program dedicated to a specific hardware and with *ad hoc* functions for the application domain consumes most of the card development cycle. In order to issue a card application,

BNSDOCID <XP 2161892A I >

what is needed is (i) to write precise specifications, (ii) to write or re-write the basic software (akin to an operating system) for possibly multiple platforms, (iii) to develop specific functions for the application, and (iv) to verify this software before to deploy it on thousands or millions of cards. This process is time-consuming and costly. Since defining specifications for products that will be available long afterwards is risky, this process is a difficulty for the creation of new markets. As it requires a long time it also severely limits the ability of a card issuer to deploy rapidly new applications in accordance with the market needs.

In order to cope with these market needs (to simplify, a reduced "time-to-market" and flexibility for card applications) new generations of smart cards (called *open* smart cards) have emerged during the last two years. Most notable efforts towards such smart card systems are Java Card [12, 13], MultOS [5], and Smart Card for Windows [6] which provide application developers an opportunity to create applications on a common base of code. They contain a platform for the dynamic (*i.e.* on demand) storage and the execution of downloaded executable content, which is based on a virtual machine for portability across multiple smart card microcontrollers and for security reasons.

While these new smart cards bring solutions regarding the market needs, they also introduce new problems for smart card makers. They provide solutions for card application developers by enabling them to program in high-level languages, on a common base of software (an abstract machine and application programming interfaces) which isolates their code from specific hardware. In that sense, they can reduce drastically the time to get new applications to market. They also tend to support both the flexibility and the evolution of applications by enabling the downloading of executable content in already deployed smart cards. This later characteristic requires more sophistication in term of security techniques since any program could potentially damage or misappropriate the whole card system. This paper deals with this issue as summarized in the next section.

1.3 Outline of this Paper

Ensuring that a program cannot damage a system consists in denying it access to other memory areas than those reserved for its execution and data (containment or "sandboxing"). Containment relies on access controls to memory areas. It would be better to associate containment with a protection mechanism that also verifies that every instruction accesses data with respect to their types. In fact, a more fine-grained protection consists in verifying that every instruction that accesses to a memory area is compliant with the definition of the data stored in this area (*i.e.* its type, or its class in an object-oriented system). This later technique has been popularized in the Java language [4].

In the field of open smart cards, this security issue is generally addressed by the use of three techniques. Current solutions for smart cards rely on three techniques. For Java Card, either an off-card verifier-converter performs a static

BNSDOCID: <XP 2181892A I >

verification of type-safety, or a defensive virtual machine performs the verification at runtime. For other types of open smart cards, no type-checking is carried out and the trust is only based on the containment of applications. For traditional smart cards (with all the code burned in their ROM), these techniques are not used because the confidence in the programs is based on the fact that they have been tested and verified before the delivery of the cards.

In this paper, we propose a new architecture to build smart card code that is based on a typed intermediate language called FACADE. It aims at providing a coherent system for card software engineering enabling both, the production of efficient traditional smart cards, and the building of type-safe downloadable content for open smart cards.

Section 2 debates the security schemes used in open smart cards before detailing the FACADE architecture. In Section 3 we focus on the verification process and on the precise static semantics of FACADE. A formal model using the B language exhibits the type-safety properties to be checked. Finally, Section 4 summarizes the paper and gives our forthcoming future work.

2 FACADE Architecture

2.1 Language Related Work

The current open smart cards provide programmers with the ability to download programs dynamically. This characteristic aims at making smart card systems more convenient in two ways. On the one hand, they support the use of high-level languages (like Java with Java card, or Visual Basic with Smart Card for Windows) in spite of smart card constraints. On the other hand, they guarantee that the programs loaded in a smart card are not able to compromise the security of the other programs.

Smart Card Compilation and Loading Process Though current smart cards are able to run programs written in high-level languages, they still have drastic limitations. Generally, the compilation of programs expressed in high-level languages produces a code unsuited to the available hardware. Two solutions have been used in the existing products.

The first solution consists in defining programming languages dedicated to smart cards. These languages tend to be close to those used on workstations but, they force programmers to take into account some specific aspects of smart card microcontrollers. For example, the language MEL has been specifically defined to program the MultOS card operating system [5]. For convenience, a domain specific compiler can generate MEL code from C.

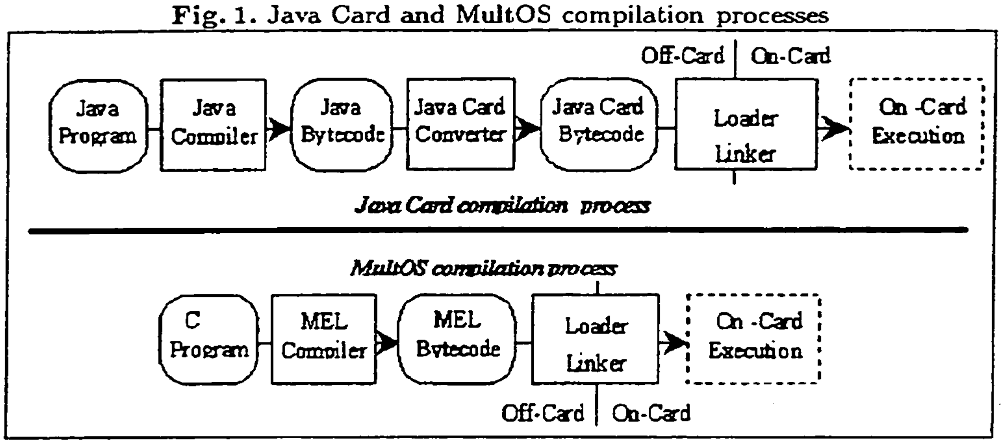
Another solution consists in converting the code produced by a traditional compiler into a specific code adapted to the smart card constraints. For example, the Java class file format is too heavy to be loaded in Java Card. Thus, the class files produced by compilers will be treated by a specific converter in order to

BNSDOCID: <XP 2181892A I >

get a compressed version of class files [14]. Smart Card for Windows is another example of the same technique.

Figure 1 presents the process of compilation and loading associated with these two approaches.

Fig. 1. Java Card and MultOS compilation processes

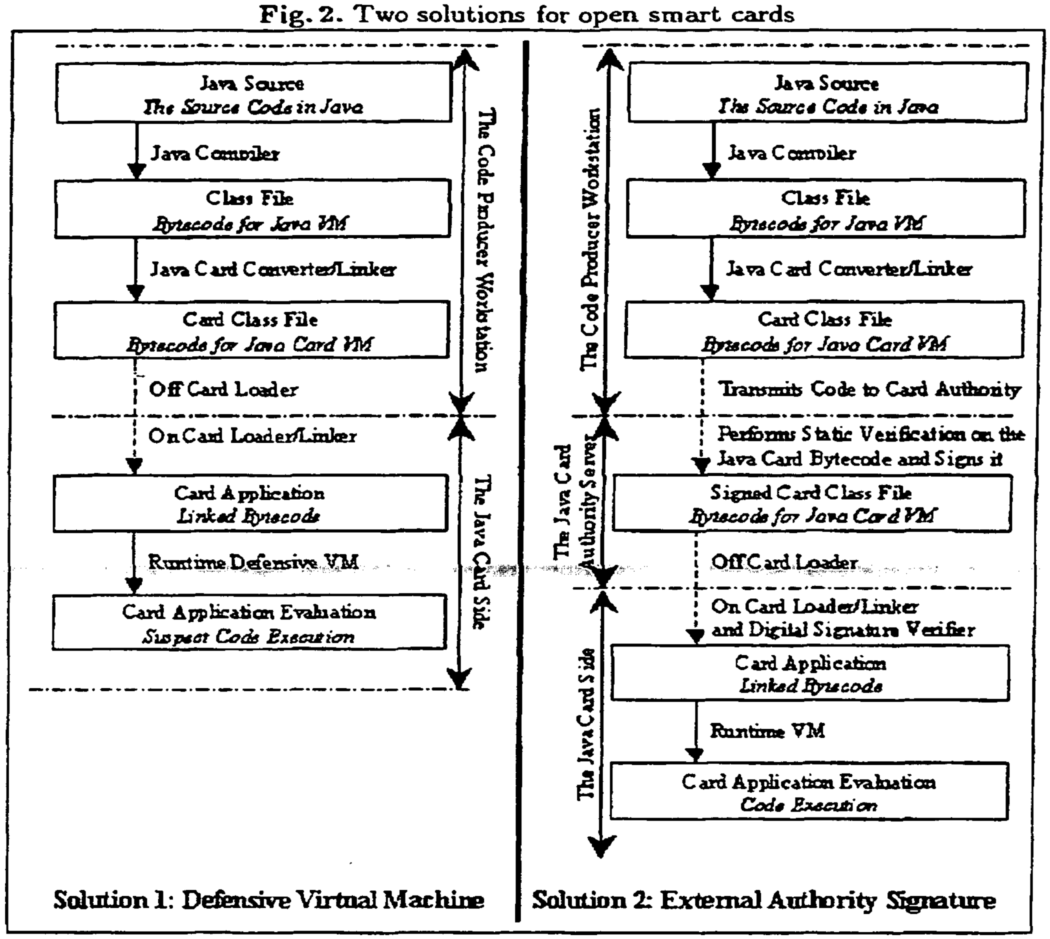


Code Downloading and Smart Card Security Currently, the Java Card framework does not support Java-like dynamic class loading. The main problem is that a smart card environment is too small for running the Java class file verifier. Open card architectures hence propose a downloading framework with reduced flexibility, in which the download unit is the application (or package in Java) rather than a single class file. In addition, new frameworks for program distribution and downloading have been proposed. The two major proposals are outlined in Figure 2.

The first solution is known as *defensive virtual machine*. All safety checks are performed at run-time, hence replacing the pro-active static bytecode verification by a defensive run-time bytecode verification. For instance, before performing a write operation to a given memory area, the virtual machine checks the type and access rights associated with the data located in that target area. In Java Card, the checks are mostly related to typing; in MultOS, the checks are related to access right checks, since security is based on application containment.

Defensive virtual machines have two major drawbacks. First, the number of run-time checks to be performed severely hinders their performance; then, some additional data (such as typing information) needs to be stored, which increases the memory requirements. As a consequence, this technique can only be applied to small programs, and this drastically reduces its flexibility and usefulness.

Fig. 2. Two solutions for open smart cards



The second solution consists in performing the security checks outside the smart card. Before to download a program onto a smart card, it is first transmitted to a trusted third-party (usually the card issuer), which checks the program and certifies it. When the program is downloaded, it is accompanied with its certificate. Before to allow this program to run, the smart card checks the validity of the certificate. This solution is currently the one that is proposed by the major card issuers. However, it has a severe drawback: the whole security of the smart card system relies on the security of the certification scheme. This solution is satisfactory for local deployments, but it can be unsatisfactory for long-term deployment of large scale applications. In fact, a central point of trust (the certification authority) creates a single point of failure that can compromise all the security architecture in case of attack. Also, in some cases, this centralization is not well suited to application needs in which the various participants are not able to trust a same authority. Moreover, the implementation and deployment

NSDOCID: <XP 2161892A I >

of a certification infrastructure is difficult and requires costly operations like monitoring and maintenance. For these reasons, we argue that it would be far preferable to design a solution in which the security of the smart card system relies on the smart card system rather than on a certification scheme.

Another Approach: FACADE A recent research work [10] proposes to adapt the Proof Carrying Code [8] technique as a mean for verifying on-card the type-safety of Java Card programs. This technique seems very promising, but the complexity of the on-card processing for a full type-checking makes it still difficult to implement on current smart cards. We propose to solve this problem by introducing a typed intermediate language called FACADE in which the type-safety can be verified on-card. As, this language is intended to be a target platform for multiple high-level languages such as Java or Visual Basic, it also solves some interoperability problems between new open smart card systems. The FACADE language is also a part of a framework for producing card programs. Inside this framework, it has two main properties: (1) it is designed specifically for an implementation on reduced feature devices, and (2) it is designed to enable an efficient static checking.

2.2 The FACADE System

The FACADE language is targeted toward very small platforms, but it is not specifically targeted toward open smart cards. The advantage of using such an intermediate language is that it can be used for all kinds of smart cards and small devices. The language is secure enough to be downloaded into an open smart card, but it can also be used to produce optimized native code, to be included in the ROM of a smart card. The main application of such an idea would be to use an high-end open smart card to prototype an application, and then to produce an optimized dedicated code in order to obtain smaller (and cheaper) cards for mass production.

In our system, the FACADE typed intermediate language is a central element, as shown in Figure 3. Programs written in various source languages are first fed into a language-specific *off-card front-end* which does lexical analysis, parsing, type-checking, and compilation into specific representations. Then, it is translated into the FACADE intermediate format. The *off-card middle end* does conventional optimizations. It also generates the elements that will be necessary to prove the type-safety of the program when it will be loaded into the smart card. At this step, the FACADE code may be used in two different directions:

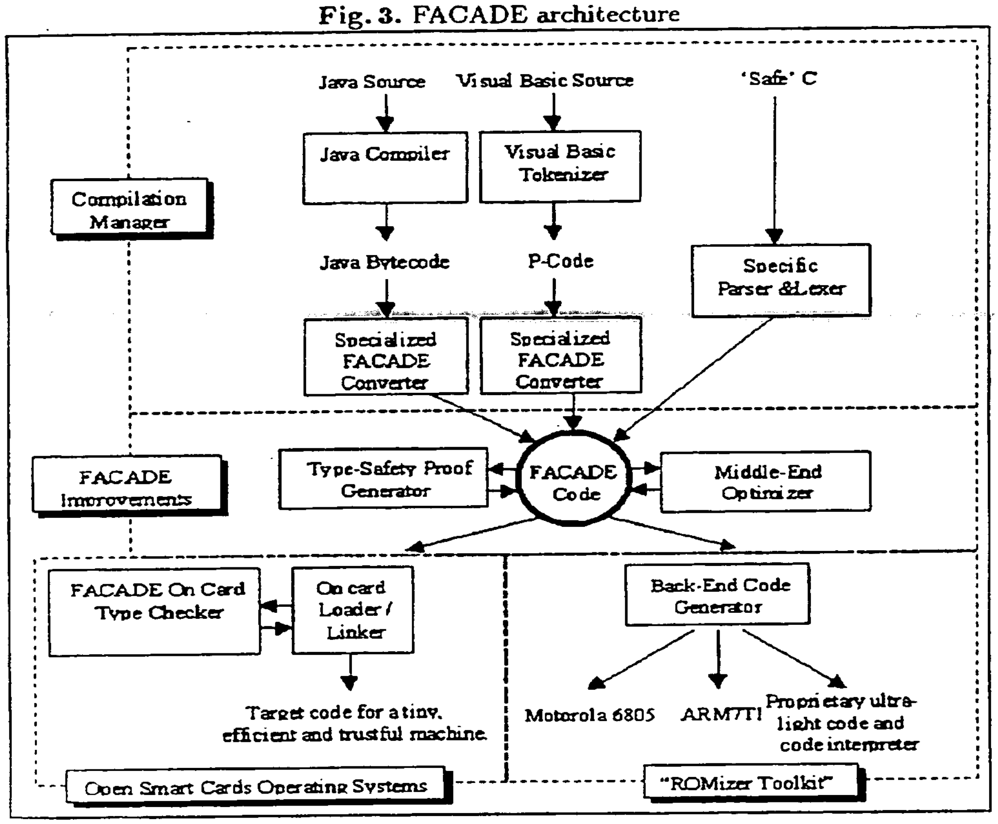
1. The *on-card back end* for open smart cards verifies the type-safety of the program at loading time, and generates a dedicated code for the target hardware or software machine.
2. The *off-card back end* for *ad hoc* smart cards used the FACADE code as the source code for producing a complete card code dedicated to be burned in ROM thanks to the usage of a ROMizer[1]

[1] A ROMizer is a software used to produce ROM binary images of programs.

BNSDOCID <XP 2181892A I >

All the production tools are deliberately made independent of each other so that they may support different source languages and different target platforms. This paper only deals with the proof generator and verifier tools. Other components of this architecture are currently investigated by ongoing experimental research works.

Fig. 3. FACADE architecture



Using a common intermediate language to share compiler infrastructure is not a new idea. Many compilers have used or use a common intermediate format for multiple source languages (*e.g.*, GNU gcc, Eiffel, Pascal, *etc.*). For a long time, Eiffel has used the C language as its intermediate language. For more advanced systems, specific languages have been defined. For instance, the FLINT architecture [11] is based on a typed intermediate format that supports higher-order functions and an advanced polymorphic type system. However, none of these languages is appropriate for small platforms such as smart cards.

The main characteristic of FACADE is its ability to prove the type-safety of FACADE code using small hardware like a smart card. This feature is very

BNSDOCID: <XP 2151992A I >

important in order to use the language as a target for the compilation of the Java language, whose security relies heavily on type-checking. Here, instead of trying to fit a standard Java type-checker on a smart card (as done in [9]), we have chosen to design a language which caters to the very specific needs of smart cards. In the following sections, we focus on the way in which type-checking can be performed on the FACADE language on a smart card.

2.3 Principle of Type-Checking: Type Inference

Program checks are usually based on data flow analysis. This analysis aims at determining the type of the variables at each program point (*pp*). The type of some variables changes during the execution of the processing because they are temporarily used for different computations.

The Hierarchy of Types In the FACADE architecture, all information are associated with types. FACADE is an object-oriented intermediate language. Thus, types are commonly defined by classes. The hierarchy of classes is an extensible data structure. Indeed, when new applications are loaded, they add their own classes to those already recorded in the card system. These extensions are done in a tree-based model. Each class has one and only one super class. Moreover, one particular class: *CardObject* is the top of the hierarchy. Thus, directly or indirectly, any class extends *CardObject*, which is the root of the inheritance graph.

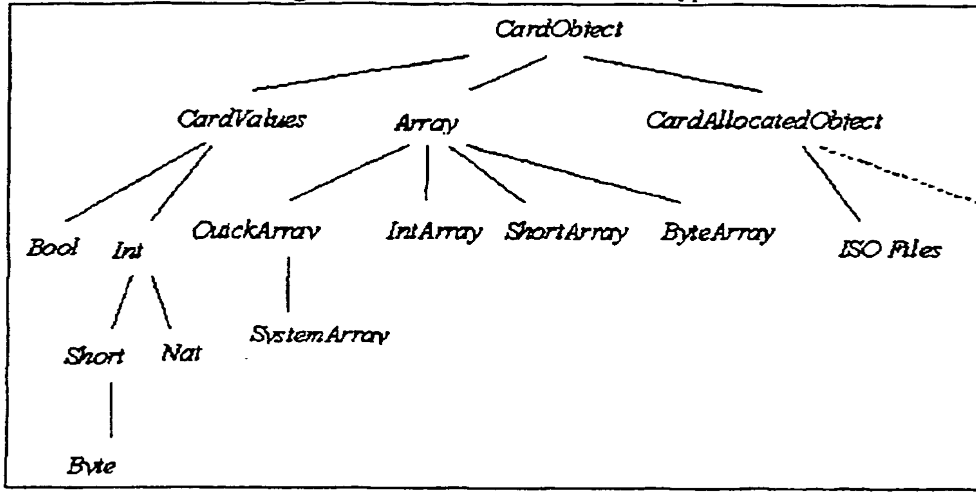
For the data flow analysis, we define a semi-lattice as a tuple $L = (V, \subseteq, \cap)$ where V is the set of the class types, $\subseteq$ is a partial order defined over V, and $\cap$ is a binary operation defined over V. Two elements $i, j \in V$ are incomparable, if neither $i \subseteq j$ nor $j \subseteq i$. We say that if $i, j \in V$, j covers i if $i \subset j$ and there is no k such that $i \subset k \subset j$. In the diagram of an ordered set $(V, \subseteq)$, two elements $i, j \in V$ are directly connected if one covers the other. The Figure 4 presents the ordered set of FACADE types. The greatest element *CardObject* of V is called the top element of V and can be written $\top$.

Type-Safe Operations A program is made of a sequence of elementary operations. Each one uses some number of input data and modifies some number of output data. In FACADE, there are only five distinct elementary operations. Following, for each of them we informally provide (i) their operational semantics, and (ii) their static semantics.

1. **Return** *VarRes*
   (i) This operation ends the execution of the processing and returns the value of the variable *VarRes*.
   (ii) This operation is legitimate if the type of the returned variable (*VarRes*) is a subtype or equals of/to the return type declared in the method signature.

BNSDOCID: <XP 2161892A I >

Fig. 4. Ordered set of FACADE types



2. `Jump` *LabelId*
   (i) This operation is an unconditional jump to another operation of the program marked with the label *labelId* defined in a label table.
   (ii) This operation is defined only if the label *labelId* exists and the value of the associated *pp* is valid.
3. `Jumpif` *Var LabelId*
   (i) This operation is a conditional jump. The *Var* variable is of type *Bool* which determines if the jump must take place or if it must be ignored.
   (ii) If the label *labelId* exists and the variable *Var* is of type *Bool*, this operation is correct.
4. `Jumplist` *Var nbcase* $\{LabelId1, LabelId2 \ldots\}$
   (i) This conditional jump reaches the label whose sequence number, in the list of the labels, is equal to the variable *Var*. If the variable *Var* is negative or if *Var* is a number greater than the cardinal of the label list (defined by the immediate value *nbcase*), the operation does not perform any jump but increments the program point.
   (ii) If each label $LabelId_x$ exists, if the number of labels is less or equal than *nbcase*, and if the variable *Var* is of type *Int*, then the operation can be carried out.
5. `Invoke` *VarRes Var methodId* $\{tabVar1, tabVar2 \ldots\}$
   (i) This operation executes the method *methodId* on the variable *Var*, with the parameters $tabVar1, tabVar2 \ldots$ *VarRes* is the variable that contains the value returned by the method call. In fact, this operation of the intermediate language FACADE could be translated in the card, according to the nature of the method *methodId*, either like a jump into another procedure, or like a set of elementary operations of the target machine.
   (ii) The call of a method is type-safe if first (a) the method *methodId* is

declared in the class (the type) of *Var*, and second (b) the types of the variables *tabVar*1, *tabVar*2 ... are included in those awaited by the method *methodId* of the class of *Var*, and finally (c) the type of the variable *VarRes* is that turned over by the method *methodId*.

An elementary FACADE operation is considered type-safe if the inferred types are conform with those defined by the static semantics. For instance, let `i` and `j` be respectively of types *Int* and *IntArray*; an incorrect operation can be `Invoke j,i,add,j`. However, the addition method of the *Int* class takes an integer value as the second parameter. As *IntArray* is not a subtype of *Int*, such operation is illegal. In this other example, let *i* a variable of the type *Short*, the following operation is correct: `Jumplist i, 4, (l1, l2, l3, l4)`, because the type *Short* is a subtype of *Int*.

In fact, the variable type checks may appear to be simple if the type of a variable do not change during the life span of the method. Unfortunately, it is not the case for all kinds of variables. We distinguish two kinds of variables, those which have an invariable type (*e.g.*, the attributes of an object), and those which have a variable type (*e.g*, the variables used for temporary storage during the evaluation of complex expressions). In the first case we speak about *Local* and in the second, we use the term *Tmp*.

Type-Safe Verifier and Smart Card Constraints A type verifier proves that each instruction contained in a program will be executed with values having the types expected by the instruction. The type verifier reads the program and notes for each elementary operation, the produced types and the consumed types.

Generally the execution of a program is not sequential. This is why a traditional verifier follows the arborescent path of each case of the program execution. The computation of this tree structure establishes with certainty the types of the variables consumed and modified by the elementary operations. When various paths end in a same program point, the safe-type verifier determines a single type for each *Tmp* variable.

The fixpoint search is the most complex aspect of the type-checking algorithm. As shown previously, it requires the production of a complex data structure, and thus the execution of a complex recursive processing. The smart card cannot perform this recursive processing, because of its hardware limitations. Therefore, no smart card proposes an operating system performing at load time type-safety checks. And if the type-safety checks are performed at runtime, they slow the virtual machine.

A Two-Step Solution Some studies propose to distribute the type-checking of a downloadable code between a code-producer and a code-receiver. The most outstanding works in this field are Proof Carrying Code [8] (PCC) and Typed Assembly Language [7] (TAL). The main idea of these works is to generate by the code-producer a proof of the program safety. The code is transmitted with this proof to the receiver. The code-receiver checks the program using the proof.

The interest of this technique is that the proof verification is much easier than the proof production [10].

In our case, the proof is the result of the type combination for a given label. Like TAL, we propose to provide, with the transmitted program, a list of labels with a state of the *Tmp* variables for each of them *i.e.* for each program point which can be reached by a jump. The receiver performs a sequential analysis of the code. When an operation is marked by a label, the receiver checks if its current inferred types are lower or equal (relation $\subseteq$ of the types hierarchy) to those defined by the proof. Moreover, when the receiver checks a kind of jump, it makes sure that its inferred types are lower or equal to those defined for the label. A program is refused if a control fails.

3 Type-Safety Verification

In the FACADE approach, the verification process is split in two parts. The resource consuming algorithm (*i.e.* the label generator) is done on the terminal side (code-producer) while the verification of the labels and the type inference between two labels is done on the card side (code-receiver). The first algorithm is a traditional type reconstruction and verification of all the possible execution paths. For each label (*i.e.* each point that can be reached by a jump operation), the joint operation is computed for the untyped local variables. The label table is transmitted to the card with the FACADE code. At loading time, the card sequentially computes the inference and compares it with the label table. In case of divergence the card refuses the code.

The verification process must ensure that every execution will be safe. It means that starting in a safe state each operation will lead the system in another safe state. For that purpose, a transition system describing a set of constraints is constructed. It defines a correct state for each operation (preconditions) and how the system state evolves (post-conditions). Preconditions and post-conditions define the static semantics of FACADE. In particular, it must be ensured that:

- all instructions have their arguments with the right type before execution,
- all instructions transform correctly the local variables for the next *pp*.

3.1 Static Semantics of FACADE

The system maintains some tables: the class descriptors (*classDsc*) and the method descriptors (*methodDsc*). The class descriptors table contains the definitions of the classes present in the card and the class hierarchy. The method descriptors table contains for every class the signatures of the already checked methods. Once a method has been accepted by the verifier, it is added in the method descriptors.

We briefly introduce our notation. Programs are treated as partial maps from addresses to instructions. If *Pgm* is a map, $Dom(Pgm)$ is the domain of *Pgm*. $\forall pp \in Dom(Pgm), Pgm_{pp}$ is the value of *Pgm* at program point *pp*,

SDOCID <XP 2161892A I >

which is written $Pgm[pp \rightarrow v]$. A program P is a map from program points to instructions. The model of our system is represented by a tuple: $< pp, Tmp >$ where pp denotes a program point (or program counter), and Tmp the set of untyped local variables. The types of the typed local variables noted L are never changed, and therefore are not part of the program state. The vector of map T contains static information about the local variables. The vector T assigns types for the Tmp variables at program point pp such that $\forall i, (Tmp[i] : T_{pp}[i])$. A program is well typed if there exist T that satisfies: $T \vdash P$.

A program is correct if the initial conditions are satisfied and if every instruction in the program is well typed according to their static constraints:

$$\frac{\begin{array}{c} Dom(T_1) = \{\} \\ \forall i \in Dom(P), (T, i \vdash P) \end{array}}{T \vdash P}$$

Following Figure 5 we give a formal definition for the control flow instructions (`Return`, `Jump`, `JumpIf`, and `JumpList`) and for the invocation instruction (`Invoke`). There are two static semantics for the `Invoke` instruction depending on the kind of variable used for $VarRes$.

3.2 Off-Card Label Generation

During this phase, we can construct the type information for local variables for each program point using a traditional type inference algorithm (for example, the Dwyer's algorithm [2]) and then, we calculate for every label the value of T for every path. An inference point (ip) is a pair made of a program point associated with the T type table for that program point (written T_{extern}). They are collected in a table – the $labelDsc$ table – which is sent to the smart card. When a new method has been loaded, it is verified that the label descriptors only reference valid addresses: $\forall i, (i \in labelDsc[pp], i \in Dom(P))$. Finally, we can formally define the label descriptors by:

$$\begin{array}{c} ip \in labelDsc, ip = (pp, T_{extern}) \\ pp \in Dom(P) \\ T_{extern} = T_{pp} \end{array}$$

Consider the following example (see Table 1), this piece of FACADE code (we call it program P) uses two variables i and v in order to compute a loop from 0 to 100.

Using a fixpoint algorithm, the inference procedure checks all the paths of the tree. In this example, it needs eight steps as illustrated Table 2.

At step 1, i and v are Tmp variables for which the types are unknown before the operation. The type of the return variable for the method `likeIt` from the class Int is Int. Thus, at step 2, i is assigned the type Int. At step 4, a first choice is made by jumping at pp 3, the other choice is verified at step 8. At step 6, the state is different than the step 3 (for the same pp) thus, we have not

BNSDOCID: <XP 2161892A I >

Fig. 5. Static semantics of FACADE control flow and invocation instructions

$$\frac{\begin{array}{c} P[i] = \texttt{Return}\ VarRes \\ i \neq Card(P) \\ VarRes : methodDsc[ThisMethodId][ReturnType] \\ i+1 \in Dom(P) \end{array}}{T, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{Jump}\ LabelId \\ LabelId \in Dom(P) \end{array}}{T, i \vdash P} \qquad \frac{\begin{array}{c} P[i] = \texttt{JumpIf}\ Var\ LabelId \\ Var : Bool \\ LabelId \in Dom(P) \\ i+1 \in Dom(P) \end{array}}{T, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{JumpList}\ Var\ nbcase\ Label \\ Var : Int \\ nbcase : Nat \\ nbcase \geq Card(Label) \\ \forall pp, (pp \in Label \Rightarrow pp \in Dom(P)) \\ i+1 \in Dom(P) \end{array}}{T, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{Invoke}\ VarRes\ Var\ MethId\ tabVar \\ MethId : methodDsc_{classDsc(Var)} \\ VarRes \in T \\ T_{i+1} = T_i[VarRes \rightarrow methodDsc[MethodId][ReturnType]] \\ \forall i, (i \in 1..methodDsc[MethodId][nbvar] \Rightarrow tabVar[i] : methodDsc[MethodId][i]) \\ i+1 \in Dom(P) \end{array}}{T, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{Invoke}\ VarRes\ Var\ MethId\ tabVar \\ MethId : methodDsc_{classDsc(Var)} \\ T_{i+1} = T_i \\ VarRes \in L \\ VarRes : methodDsc[MethodId][ReturnType] \\ \forall i, (i \in 1..methodDsc[MethodId][nbvar] \Rightarrow tabVar[i] : methodDsc[MethodId][i]) \\ i+1 \in Dom(P) \end{array}}{T, i \vdash P}$$

BNSDOCID <XP 2161892A I >

Table 1. A sample FACADE program

pp	Label	Instruction	Comment
1		`Invoke i, 0, likeIt`	i is set to 0
2		`Jump label0`	go to the test at `Label0`
3	`Label1`	`Invoke i, i, add, 1`	increment `i`
4	`Label0`	`Invoke v, i, LtOrEq, 100`	test if `i` less or equal than 100
5		`JumpIf v, label1`	if TRUE go to `Label1`
6		`Return void`	otherwise, return

Table 2. Off-card type inference procedure of the sample FACADE program

Step	*pp*	Label	Instruction	$T[i, v]$
1	1		`Invoke i, 0, likeIt`	$(\top, \top)$
2	2		`Jump Label0`	$(Int, \top)$
3	4	`Label0`	`Invoke v, i, LtOrEq, 100`	$(Int, \top)$
4	5		`Jumpif v, label1`	$(Int, Bool)$
5	3	`Label1`	`Invoke i, i, add, 1`	$(Int, Bool)$
6	4	`Label0`	`Invoke v, i, LtOrEq, 100`	$(Int, Bool)$
7	5		`JumpIf v, label1`	$(Int, Bool)$
8	6		`Return void`	$(Int, Bool)$

reached a fixpoint. At step 7, states are identical, meaning that a fixpoint has been reached. Now, the pending path memorized at step 4 can be checked.

For the *pp* 3, we have to make the combination between the two states $T_4[i, v] = (Int, Bool)$ (at step 6) and $T_4'[i, v] = (Int, \top)$ (at step 3). Using our semi-lattice, the greatest lower bound of $T_4[i, v]$ and $T_4'[i, v]$ is $T_{extern}[i, v] = (Int, \top)$. Our *labelDsc* is made of two *ip*: $(4, [Int, \top])$ and $(3, [Int, Bool])$.

In order to optimize the size of the data sent to the smart card, the *labelDsc* table can be reduced. In fact, by verifying the definition-use paths, one can remark that the information on the variable v at step 6 is useless because v is always defined before any use. Thus, it is possible to simplify the *labelDsc* as shown Table 3.

Table 3. Simplification of the label descriptors table (*labelDsc*)

ip	*pp*	$T_{extern}[i, v]$
0	4	$(Int, \top)$
1	3	$(Int, \top)$

3.3 On-Card Type Inference and Verification

The on-card FACADE verifier uses the off-card-generated label descriptors in order to reduce its footprint in memory as well as the time needed by this process. After receiving the application and its *labelDsc*, the verifier checks the conformity of the label descriptors. Then, it begins the verification process by running sequentially through the code.

The procedure is the following (see Table 4): for each instruction the verifier checks if the program point is referenced into the label descriptor. In such a case, the table T_{extern} replaces T_c otherwise, the algorithm uses T_c. It applies the static semantics to verify the correctness of the code. After an unconditional Jump, the following instruction must have necessarily a label, and then it replaces T_c by T_{extern}.

Table 4. On-card verification procedure of the sample FACADE program

Step	*pp*	Instruction	$T_{extern}[i, v]$	$T_c[i, v]$
1	1	`Invoke i, 0, likeIt`		$(\top, \top)$
2	2	`Jump label0`		$(Int, \top)$
3	3	`Invoke i, i, add, 1`	$(Int, \top)$	$(Int, \top)$
4	4	`Invoke v, i, LtOrEq, 100`	$(Int, \top)$	$(Int, \top)$
5	5	`JumpIf v, label1`		$(Int, Bool)$
6	6	`Return void`		$(Int, Bool)$

At step 2, before the Jump, T_c has been inferred as $[Int, \top]$. At step 3, the program pointer is included in the label descriptors thus, T_c is overloaded by the external table T_{extern}. At step 4, the transfer function associated with the method `LtOrEq` applied on an *Int* changes the type of the return parameter to a *Bool*. At step 5, the precondition to a `JumpIf` is to have a *Bool* value as parameter, which is satisfied.

3.4 Status

The complexity of the verification process is $\theta(n)$ since the algorithm verifies sequentially the instructions. Furthermore, the memory usage is reduced because the algorithm only needs the variables types of the current *pp*. If the type information and the applet have been modified, the verifier always refuses the code.

In our architecture, the card security relies mainly on the verification process. Therefore, the design of the type-checker cannot suffer any error. In a recent paper [1], we specified a formal model of a Java Card bytecode verifier and we provided the proof of its correct implementation. As the Java Card and the FACADE data flow verification are closed, we expect to achieve the formal proof

BNSDOCID: <XP 2161862A I >

of the FACADE verifier in a near future. This work is part of a Ph.D. program, and the results will be presented in a future paper.

4 Conclusions and Future Works

The FACADE project is an academic research effort supported by the Gemplus Research Lab. It investigates the issue of software production in the field of smart cards. We have introduced the specific domain of smart cards. We have reviewed the programming techniques used for the production of traditional cards and open smart cards. Then, we have presented the FACADE framework for producing smart cards programs.

This framework is based on a typed intermediate language dedicated to smart cards. The first advantage of this single intermediate format is that it enables migration paths between open and traditional cards, from a variety of source languages towards a variety of smart card runtime environments and hardware platforms. There are also other advantages in using a typed intermediate language. First, a rigorous type system can be used to verify the safety of a program. Second, it is possible to abstract the programmers from a single source language if programs of different surface languages share the same runtime system based on a uniform type system. Finally, type safe languages have been shown to support fully optimizing code generation and can efficiently implement security extensions (access control lists or capabilities).

This paper focused on the type-safety verification process. We have shown that it is split in two parts. The off-card part is resource consuming but it generates important information (the labels) in order to minimize the second part of the verification process. By this mean, the on-card part is able to perform the verification with the reduced features of a smart card. For the moment, we are working on the complete B model of the verifier. The expected result is a proof of the correctness of the verification process.

Future works are experimental researches on the execution environment of FACADE programs. They include the generation of target code from the FACADE language and the implementation of the runtime system libraries. This work will give us metrics on code size and efficiency, but also an implementation of the operational semantics. In order to prove also the correctness of the execution process, we intend to develop a formal model of the dynamic semantics.

Acknowledgments

We first thank Éric Vétillard for providing us with material to write some parts of this paper, and also Patrick Biget for his helpful comments on this paper. But, Éric must undoubtedly be acknowledged for his careful reading of the paper and his insightful comments which helped us to improve the paper greatly.

BNSDOCID <XP 2161892A _ I >

References

1. CASSET, L. How to formally specify the Java bytecode semantics using the B method. In *ECOOP Workshop, Formal Techniques for Java Programs* (Lisbon, Portugal, June 1999).
2. DWYER, M. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*. PhD thesis, University of Massachusetts, Sept. 1995. [http://www.cis.ksu.edu/~dwyer/papers/thesis.ps].
3. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *International Standard ISO/IEC 7816: Integrated circuit(s) cards with contacts, parts 1 to 9*, 1987-1998.
4. LINDHOM, T., AND YELLIN, F. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Sept. 1996.
5. MAOSCO LTD. *"MultOS" Web site*. [http://www.multos.com/].
6. MICROSOFT CORP. *"Smart Card for Windows" Web site*. [http://www.microsoft.com/windowsce/smartcard/].
7. MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From System F to Typed Assembly Language. In *25th Symposium on Principles of Programming Languages* (San Diego, CA, USA, Jan. 1998). http://simon.cs.cornell.edu/Info/People/jgm/papers/tal.ps.
8. NECULA, G. C. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France, Jan. 1997). [http://www.cs.cmu.edu/~necula/popl97.ps.gz].
9. ROSE, E. Towards Secure Bytecode Verification on a Java Card. Master's thesis, University of Copenhagen, Sept. 1998. [http://www.ens-lyon.fr/~evarose/speciale.ps.gz].
10. ROSE, E., AND ROSE, K. H. Lightweight Bytecode Verification. In *Formal Underpinnings of Java, OOPSLA '98 Workshop* (Vancouver, Canada, Oct. 1998). [http://www-dse.doc.ic.ac.uk/~sue/oopsla/rose.f.ps].
11. SHAO, Z. Typed Common Intermediate Format. In *USENIX Conference on Domain-Specific Languages* (Barbara, CA, USA, Oct. 1997). [http://flint.cs.yale.edu/flint/publications/tcif.html].
12. SUN MICROSYSTEMS, INC. *Java Card 2.0 Language Subset and Virtual Machine Specification, Programming Concepts, and Application Programming Interfaces*, Oct. 1997. [http://java.sun.com/products/javacard/].
13. SUN MICROSYSTEMS, INC. *Java Card 2.1 Virtual Machine, Runtime Environment, and Application Programming Interface Specifications*, Public Review ed., Feb. 1999. [http://java.sun.com/products/javacard/javacard21.html].
14. SUN MICROSYSTEMS, INC. *Java Card 2.1 Virtual Machine Specification*, Draft 2 ed., Feb. 1999. [http://java.sun.com/products/javacard/JCVMSpec.pdf].

Defensive Java Virtual Machine Version 0.5 alpha Release

May 13, 1997 XP-002161893

P.D. 13-05-1997
P. 1-14+23-38-49
61-62+93-94+131-133

Over the last several months Computational Logic, Inc. (CLI) in collaboration with Schlumberger Electronic Transactions and JavaSoft (see "Where is JavaSoft going with its model" in the original Javasoft announcement) has been building a formal model of a subset of the Java Virtual Machine (JVM). The model has been built using ACL2, a mathematical logic based on Common Lisp. The result can serve as the basis for rigorous, formal analysis of the JVM and JVM (bytecode) programs. Because models written in ACL2 can be executed, the formal JVM model can run programs within the subset of the JVM supported.

The model is called the "defensive" JVM (or dJVM) because it includes sufficient run-time checks to assure type-safe execution (or at least to detect and prevent any unsafe execution). In the standard JVM these checks are not present; type safety is dependent an unstated property of the bytecode verifier and the JVM.

The first phase of the CLI effort has been to build an initial model of a significant portion of the Java Card subset of the JVM. This includes object creation, field access, and method invocation. This phase has been completed, and CLI and its collaborators are making the "alpha" release of the initial model publicly available for external review and comment. This snapshot demonstrates one approach to formalizing and clarifying the JVM specification.

Availability

The draft formal description of the model is available. Please mail any comments to Richard Cohen.

- HTML version.
- The full report can be downloaded as a postscript or pdf file. Each is approximately 2MB.

  NOTE: The HTML version was generated mechanically from LaTeX sources generated mechanically from the ACL2 source files. While this has produced a readable hyper-linked rendition, there are a number of minor formatting glitches. For the authoritative documentation see the report.

The model is available as gzipped or compressed TAR files which include an executable image and the dJVM 0.5 *User's Guide* in PostScript and PDF formats:

- Gzipped for Solaris 2.5 on SPARC architecture.(31 megabytes)
- Compressed for Solaris 2.5 on SPARC architecture.(38 megabytes)
- Gzipped for Solaris 2.5 on x86 architecture.(26 megabytes)
- Compressed for Solaris 2.5 on x86 architecture.(37 megabytes)

  WARNING: The current executable images are very large; they are approximately 70MB each when expanded. The gzipped versions are about 30 MB. The image includes the executable dJVM model as well as the complete ACL2 theorem prover, compiler, and all of the definitions and theorems of the dJVM model.

The *User's Guide* briefly explains how to run the model and is available as a postscript or pdf file. The ACL2 source files comprising the model are also available as a gzipped tar file (300KB) .

Java, Java Card and Solaris are trademarks of Sun Microsystems, Inc. SPARC is a trademark of SPARC International, Inc.

This page is URL http://www.cli.com/software/djvm/index.html

BNSDOCID <XP 2161893A I >

The Defensive Java Virtual Machine Specification Version 0.5

** Alpha Version **

Richard M. Cohen

Computational Logic, Inc.
1717 West 6th Street, Suite 290
Austin, Texas 78703-4795

Telephone: 512-322-9951
Email: cohen@cli.com

May 12, 1997

** Alpha 1 Release **

NSDOCID <XP 2161893A I >

Contents

BNSDOCID: <XP _ 2161893A I >

NSDOCID <XP 2161893A I >

BNSDOCID: <XP 2161893A I >

NSDOCID <XP 2161893A I >

BNSDOCID: <XP 2161993A I >

BNSDOCID <XP 2161893A I >

BNSDOCID: <XP 2161893A I >

ISDOCID <XP 2161893A I >

Preface

The *Defensive Java Virtual Machine* (dJVM) is a *defensive* version of the Java Virtual Machine (JVM). The dJVM augments the JVM with additional run-time checks to assure type-safe execution.

We define a formal model of the dJVM using the ACL2 language. Because ACL2 is an executable mathematical logic, we obtain a model which we can run, as well as one about which we can formally reason. This provides both the clarity and proof cababilities of a formal, logical model, and the ability to validate the specification against our intensions by running test cases on the model.

The JVM is an optimized version of the dJVM, which behaves correctly (i.e., as the dJVM) under proper conditions. The Java bytecode verifier is intended to assure those conditions by rejecting bytecode programs that do not satisfy the assumptions underlying the JVM optimizations. The main portion of these assumptions address the *type safety* of the bytecode program.

Warning to the Reader

This is a draft (or "alpha" release) of the dJVM 0.5 specification. It is meant as an exploration of formalizing the Java Virtual Machine, the the completeness of the current JVM documentation, and type-safe execution of the JVM. As such, the emphasis was to build an initial formal model of the core, object-oriented portion of the JVM. The model is called dJVM version 0.5 to suggest that it is only an initial step toward formally modeling the full Java Virtual Machine. The text that accompanies portions of the formal model is (yet) not a full or adequate explanation or justification of the model.

The text is sprinkled with "author's remarks." These notes are indicative that this is an incomplete report of an incomplete model. The remarks are marked in the text as:

Some of the remarks concern questions about the model or the JVM. Some are reminders of portions of the report or the model that have to be written or revised. Author's Note

BNSDOCID: <XP 2161993A I >

Acknowledgments

Thanks to Bill Young, and Mike Smith for help and support during development of the model. Thanks to Marianne Mueller and Li Gong of JavaSoft and Scott Guthery of Schlumberger Electronic Transactions for thinking this work was worth pursuing.

And, of course, thanks to the Java team for creating a pleasing language and virtual machine, which are at once both useful enough to generate net-wide excitement and interest, and clean enough to make a formal definition reasonable (and not nearly as hard as for most other practical programming languages). My main sources of information have been the *Java Language Specification* by James Gosling, Bill Joy, and Guy Steele, and the *Java Virtual Machine Specification* by Tim Lindholm and Frank Yellin. These two works have provided examples of clear, readable, quite complete explanations of a subtly-complex programming language and virtual machine. I know that considerable care and effort went into making these works available so early in Java's growth. These two works have been the main sources for building the dJVM model.

Thanks to Sun Microsystems, Inc., for permission to use extensive quotations from the *Java Language Specification* [Gosling *et al.*, 1996] and the *Java Virtual Machine Specification* [Lindholm and Yellin, 1996].

Thanks for Schlumberger, JavaSoft, and CLI for supporting this work.

This report was prepared using Perl and $\LaTeX 2_\varepsilon$, which alternately made progress a pleasure and a puzzle, and repeatedly reminded the author that programming is a rewarding, complex, exacting, and sometimes inscrutable activity.

NSDOCID <XP 2161993A I >

Chapter 1

Introduction

The Java Virtual Machine (JVM) is a key part underlying the definition and portability of Java programs. The JVM is a relatively clean and simple abstract machine, which has been realized in software (as an interpreter) and is being realized in hardware as well. The JVM has a bytecoded instruction set designed to be compact and easily interpreted in either hardware or software.

The JVM is an object-oriented machine, manipulating objects as well as numeric data. The JVM also supports multithreaded programs, although much of the semantics of concurrent programs are defined by standard classes, rather than by the JVM itself.

The JVM was designed to allow efficient, safe execution of (bytecode compiled) Java programs. One innovative technique employed to promote efficiency is the use of the bytecode verifier. The JVM does not assure type-safe execution by itself. Rather, JVM execution is intended to be type-safe when running programs that have been "accepted" by the bytecode verifier. The bytecode verifier is *intended* only to accept programs whose execution on the JVM will be type-safe. The standard JVM is defined to elide some type-safety checks based on these intended guarantees, using these guarantees to justify optimizations in the JVM definition that would not be type-safe without those guarantees.

Thus, while the JVM execution is *intended* to be type-safe when running programs accepted by the bytecode verifier, the type-safety is not obvious. The JVM is *not* guaranteed to be type-safe when running arbitrary programs.

The *defensive* JVM (dJVM) augments the JVM definition with additional run-time checks to assure type-safe execution. The definition is intended to be obviously type-safe when run on any program. There is no need for a bytecode verifier to screen programs.

1.1 How does the dJVM differ from the JVM?

The dJVM is obviously type-safe. At least the dJVM is *intended* to be obviously type-safe. In contrast, the JVM is obviously not type-safe *by itself.* The behavior of the JVM is only specified for programs that are (mostly) type-safe. For example, the behavior of the `iadd` instruction is not defined unless the top two words on the operand stack are `int` values. In

BNSDOCID <XP 2161893A I >

contrast, the behavior of `iadd` on the dJVM is completely defined.[1]

The dJVM is completely defined. Unlike the JVM, the dJVM is *completely* defined.

> If some constraint (a "must" or "must not") in an instruction description is not satisfied at run time, the behavior of the Java Virtual Machine is *undefined* [emphasis added – RMC].
> [Lindholm and Yellin, 1996, §6.1, p. 151]

The effect of every instruction in the dJVM is defined for all possible dJVM states. For every state in which the effect of a JVM instruction is defined, the corresponding dJVM instruction has the same effect on the the corresponding dJVM state. In addition, in the states for which a JVM instruction is *undefined*, the effect of the corresponding dJVM instruction in the corresponding dJVM states is to halt with an error indication.

1.2 What use is the dJVM definition?

Some things we can do with a formal dJVM model:

- Expose/resolve questions about the JVM behavior (e.g., those not answered in the Java Virtual Machine Specifications [Lindholm and Yellin, 1996]).
- Prove that the JVM is type-safe on programs accepted by the bytecode verifier — this is a standard interpreter-equivalence proof.
- Examine the model's behavior by running test cases (assuming its an executable model).

1.3 Why is this dJVM version 0.5?

The current model is the first phase of modeling the JVM in ACL2. It was intended to demonstrate that such a model could reasonably capture the key properties of the JVM, but to be a complete JVM model. It is a rough draft.

This model includes the basic object-oriented instructions and data structures. It leaves out concurrency, interfaces, arrays, and garbage collection. It leaves out floating point data. It only includes 103 of the 202 standard JVM instructions.

1.3.1 Components of JVM included in dJVM

These are the essential features of the JVM, and were included in the dJVM 0.5 model so that this first model would address these key aspects of the JVM. These features also addresses the key aspects of object creation and manipulation in this initial phase of the model.

[1] At least the behavior of `iadd` is defined in all contexts in which such a question makes sense. For example, it does not make sense to ask about the effect of executing an `iadd` instruction if the call stack is empty, because that means no method is active to execute the instruction.

NSDOCID: <XP__ 2161893A__I >

1. Representation of class objects and class instances.

   A minimal subset of standard Java classes needed to support class definition, object creation, and method execution are provided. This includes portions of the standard classes: `Object` and `Class`.

   Required instructions: `new, aconst_null`

2. Instance methods and class methods.

   Required instructions: `invokevirtual`, `invokespecial`, `invokestatic`, `return`, `areturn`, `ireturn`.

3. Instance variables and class variables.

   Required instructions: `getfield, putfield, getstatic, putstatic`.

4. Instance initialization methods (i.e., `<init>` methods).

5. Primitive types `int` and `long`.

   Some of the instructions that manipulate `int` or `long` values are included, except those related to excluded types (e.g., the `i2f` instruction, which converts an `int` integer value to a floating point value.)

   `i2l, i2s, iadd, iand, idiv, iinc, imul, ineg, ior, isub, ixor, l2i, ladd, land, ldiv, bipush, sipush`

6. Enough other instructions to run some small examples.

   Such instructions include:

   (a) some load and store instructions

   `aload, astore, iload, istore, lload, lstore`

   (b) some stack-manipulation instructions

   `pop, pop2, dup, dup2, dup_x1, dup_x2, dup2_x1, dup2_x2, swap`

   (c) some control-flow instructions

   `goto, if_acmpeq, if_acmpne, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, ifeq, ifne, iflt, ifle, ifge, ifnull, ifnonnull`

1.3.2 Important Features Left Out For Now

A number of important features of the JVM have not been included in the dJVM 0.5 model. Some of these are candidates for the first extensions to dJVM 0.5.

1. Dynamic loading of classes.

2. Class-file verifier and bytecode verifier.

   Technically the class-file verifier and bytecode verifier are not part of the JVM itself. However, they do form an important component of most practical Java implementations. As such they would be worthy extensions of the dJVM.

BNSDOCID: <XP 2181893A I >

3. Class initialization methods (i.e., `<clinit>` methods).

   They are an implicit part of the JVM, because they are invoked implicitly when a class is created; they are never invoked explicitly.

   dJVM presumes that all class objects have been constructed and initialized prior to the start of execution (i.e., effectively constructed by some primordial loading process).

4. Interfaces, and interface initialization methods.

5. Exception handling .

   Including the instructions `jsr`, `jsr_w`, `ret`, and `athrow`.

6. Multithreading.

7. Monitors and synchronization.

   The instructions `monitorenter` and `monitorexit` have been be left out, as well as the special-case of the method-invocation instructions that deals with `synchronized` methods.

8. Garbage collection or object finalization.

   Garbage collection is logically a no-op, and so can reasonably be left out of the model.

1.3.3 Non-essential Features

This section describes features of the JVM that do not pose problems for modeling or formalization. They have been left out of this initial dJVM model merely to reduce the level of effort required.

1. Arrays and all of the instructions that create or operate on arrays

2. The data type `char`

3. The data types `float` and `double`

   ACL2 does not support floating point numbers.[2] So it is unlikely that the dJVM model will ever be extended to define floating point operations in the JVM. However, since the JVM is defined to perform floating point calculations according to the IEEE Floating Point Standard, there is little advantage to such an extension.

4. 16-bit Unicode characters.

5. The instruction-modifier `wide`.

6. Some non-essential instructions.

   (a) `ldc`

   (b) Many arithmetic instructions.

[2] ACL2 does support arbitrary-precision integers (a.k.a., bignums) and rational numbers. The axiomatic description of rational arithmetic is much more tractable than that of floating point arithmetic. Further, unlike floating point numbers, rational numbers obey the normal laws of arithmetic!

NSDOCID: <XP 2161893A I >

1.3.4 Things not yet in our world

1. Native methods.

   Native methods are really extensions to the JVM. They are effectively new instructions that are executed via the method invocation mechanism, rather than via normal instruction dispatch.

   Extending the dJVM to handle native methods would require two steps:

   (a) Extend the representation of methods to allow recording information about native methods.

   (b) Extend the method invocation instructions to allow creating a call-frame for a native method.

   (c) Extend the instruction dispatcher to recognize when the current method is a native method, and invocate the ACL2 description of the method.

   Since each native method is an extension of the dJVM, each new native method will require that an additional case be handled by the instruction dispatcher. Each time a new native is added to the model, the functions or axioms describing the native method must be added to the model and the instruction-dispatcher function must be revised to handle invocation of this new native method properly.[3]

2. External system resources (e.g., file systems, networks, *etc.*)

   Eventually we want to augment our model to describe interactions with external resources (e.g., file systems and networks).

3. The `_quick` instructions. They are implementation-specific optimiziations that do not currently fit nicely into the dJVM model. The "quick" instructions anticipate an implementation based on dispatch vectors. But the current dJVM model performs dynamic lookup, rather than caching method bindings in a dispatch vector. A variant of the current model that uses dispatch vectors or other method-caching techniques could be developed.

1.4 What is a Formal Model?

The dJVM model is built in ACL2. ACL2 is a mathematical logic based on the applicative subset of Common Lisp [Steele Jr., 1984]. As such it provides a means for describing a *formal specification* of the dJVM. ACL2 also has a *theorem prover* for the logic. This theorem prover is the latest in a line of theorem provers built by Bob Boyer and J Moore. [Boyer and Moore, 1979; Boyer and Moore, 1988; Kaufmann and Moore, 1994; Kaufmann and Moore, 19
Thus we can (attempt to) prove properties of these specifications.

ACL2 allows specifications to be built using function definitions, in the manner of pure LISP and other functional programming languages. Since ACL2 is based on a subset of Common Lisp, specifications may be executed in an underlying implementation of Common Lisp. Thus we can build *executable specifications*.

[3] Because ACL2 is a first-order logic, the instruction dispatcher must be extended explicitly. ACL2 lacks the Common Lisp functions `apply` and `funcall`. Functions are not "first class objects."

BNSDOCID <XP 2161893A I >

We prefer to use the terms *formal model* and *executable model*, rather than formal specification and executable specification, because these formal artifacts are always *models* of the real computing systems they are meant to describe. Often they are *abstract models*, leaving out some level of detail in order allow formal analysis and proofs about the behavior of the model. If it is a good model of the real computing system, then the analysis of the behavior of the model should also be predictive about the behavior of the real system.

1.5 This Report as a Program

The text of this report is mechanically derived from the dJVM definition accepted and compiled by the ACL2 system. In the tradition of Knuth's literate programming, the formal program and its description were woven together, and subsequently derived portions can be processed by either a compiler, a theorem prover, or a document formatter. [Knuth, 1992; Sewel, 1989]

Thus, the program text that appears in this report is derived directly from the ASCII files used for compilation and execution of the model. This increases our confidence in the fidelity of this presentation to the executable model.

There are two versions of this report. One includes only the executable portion of the dJVM definition, and is intended to be more usable as straightforward (and somewhat more compact) explanation of the behavior of the dJVM.

A second version of this report includes the theorems and rewrite rules given to the ACL2 theorem prover as part of the model development and to support ACL2 guard verification for the model. This is not a complete rendition of the ACL2 proof script because some ACL2 directives are suppressed in the text.[4]

ACL2 requires that programs be defined in a bottom-up fashion. Each function must be defined (or axiomatized) before it can be used. The presentation in this report follows this same basic order. However, some portions of the definition that don't contribute to the exposition appear at the end of this document as appendices. These portions include the description of bounded-integer arithmetic and the definitions of macros used to build the actual instruction definition. The macros are described in the main text, but the details of their definitions are relegated to the appendices.

The File Structure of the Model

The "source code" for the ACL2 model (and this report) are divided into a number of separate files. The files form a dependency hierarchy with most files depending upon definitions and rules introduced by other files. Here's a list of the files, and roughly bottom-up order. Several low-level files that do not contribute to the exposition appear as appendices.

1. `minor-utilities.lisp` — Appendix A
2. `preliminaries.lisp` — Appendix A

[4] All of the definitions and theorems of the ACL2 proof script are included. Numerous directives enabling and disabling rewrite rules are suppressed, as are the directives for including ACL2 "books" (i.e., libraries). Of course, all of the details are available in the ACL2 source files, from which the $\LaTeX 2_\varepsilon$ sources for this document were derived.

INSDOCID: <XP 2161993A I >

3. `arith.lisp` — Appendix B
4. `primitive-values.lisp` — Chapter 4
5. `state.lisp` — Chapters 5 through 8.
6. `internal-operations.lisp` — Chapter 9
7. `frame-operation-macro` — Appendix D
8. `define-inst-macro.lisp` — Appendix C
9. `simple-instructions-a-h.lisp` — Chapter 11
10. `simple-instructions-h-i.lisp` — Chapter 11
11. `simple-instructions-j-z.lisp` — Chapter 11
12. `tableswitch.lisp` — Chapter 11
13. `new-instance.lisp` — Chapter 12
14. `invokevirtual.lisp` — Chapter 13
15. `invokespecial.lisp` — Chapter 13
16. `invokestatic.lisp` — Chapter 13
17. `method-returns.lisp` — Chapter 13
18. `getfield.lisp` — Chapter 12
19. `getstatic.lisp` — Chapter 12
20. `run-djvm.lisp` — Chapter 14
21. `initial-djvm.lisp` — Chapter 14

1.6 The Remainder of this Report

Here's a brief overview of the remainder of this report.

- Chapters 4 through 5 describe the primitive values and data types manipulated by the dJVM.
- Chapters 6 through 8 build the representation of objects, classes, the call stack, the class table, and the heap to yield the full description of the dJVM state.
- Chapter 9 defines internal operations that manipulate the dJVM state.
- Chapter 10 describes the the standard structure we will use to define dJVM instructions, and the abbreviated forms (macros) that will be used to give those definitions.
- Chapter 11 defines 98 "simple instructions" — instructions that manipulate only the operand stack and local variables without reference to the heap or the class table.

BNSDOCID: <XP 2161893A I >

- Chapters 12 and 13 describe creation of objects and invocation of methods.
- Chapter 14 describes the top-level of the dJVM interpreter and the construction and initialization of initial dJVM states.
- The appendices include some low-level definitions that support support the model.

1.7 Naming, Spelling, and Typeographic Conventions

This report documents a proof-of-concept for building a formal ACL2 model of the defensive Java Virtual Machine. Due to the fast pace of the development of this initial model, I have not been entirely consistent in the use of naming conventions. In fact, I employed several in different sections of the model. Common Lisp and ACL2 are normally not case sensitive in resolving function and variable names. The name `Calling-Frame` and the name `calling-frame` are considered to be the same. In some places I have taken advantage of that, and have used capitalization in an attempt to improve the readability of long function-names.[5]

When function definitions are presented, the name of the function is typeset in a larger and different type face to make it stand out. The same style is used for ACL2 events that define theorems, data structures, and so on.

1.7.1 Notes on ACL2, Future Extensions, *etc.*

The text notes about ACL2, future extensions of the dJVM model, the Java language, and the Java Virtual Machine. These notes are set out from the regular text in the following style.

Note on the Java Language Specification

The first public version of the Java language manual was approximately 35 pages. The second version was about 70 pages. The third (and current) version is over 800 pages. As you ask more and more specific questions, the explanation of the details just seem to multiply.

The margin note identifies the topic of the note.

1.7.2 Marginal Cross-references

The ACL2 definitions that comprise the dJVM 0.5 model appear in the text in a `fixed width` typeface. To facilitate perusal of the definitions cross-references to functions appear in the margins. For example, the definition of the function `local-method?` from page 79 appears below.

```
(defun local-method? (method-name method-sig class-decl)
  (declare (xargs :guard (and (stringp method-name)
                              (stringp method-sig)
```


!SDOCID <XP 2161893A I >

```
72                              (CLASS-DECL-P class-decl))))
   (JAVA-METHOD-BOUND? method-name
                       method-sig
                       (CLASS-DECL-METHODS class-decl)))
```

The boxes appearing in the margin are cross references for a function that appears on the corresponding line of the definition. In this case indicating that the definition of `class-decl-p` can be found on page 72, and the definition of `java-method-bound?` can be found on page 66. The function to which the cross-reference refers is printed in SMALL CAPITAL letters (e.g., as JAVA-METHOD-BOUND? and CLASS-DECL-METHODS above).

If some function names are `underlined`, *ignore the underlining. The underlining is just an indication of where the automatically generated cross-reference mechanism has broken down. The underlining should be removed in a later draft.*

Author's Note

LaTeX 2ε is supposed to put margin notes in the outer margin of the page. However, it doesn't always get it right, as you may be able to observe on this page and elsewhere in this document. Apparently the page-fill algorithm doesn't reliably communicate with the margin note placement mechanism when text is moved to start a new page. I'm not sure how to fix this.

Author's Note

[5] I'd like to use mixed-case function names throughout the presentation, but don't yet.

BNSDOCID: <XP 2161893A I >

JSDOCID: <XP _2161893A_ I >

Chapter 2

Overview of the dJVM Specification

The dJVM specification is a formal description of a defensive version of the JVM. This description of the dJVM 0.5 is complete, in the sense that all questions about the behavior of the dJVM can be answered strictly from the ACL2 definitions in this formal description.[1] However, this report does not by itself present a full explanation of the JVM or the Java language. This report is more akin to the first draft of a formal reference manual for the dJVM. It serves to answer questions about the virtual machine's behavior in detail, but does not attempt to serve as an introduction either to the Java Virtual Machine or the Java programming language. The reader should already be familiar with them, or be prepared to assimilate them from the formal definitions with the occasional accompanying brief explanations.

We recommend that the reader not already familiar with these topics have ready access to *The Java Virtual Machine Specification* [Lindholm and Yellin, 1996]. *Java Virtual Machine* [Meyer and Downing, 1997] provides an informal, but less authoritative, introduction to the JVM. Access to *The Java Language Specification* [Gosling *et al.*, 1996] may also be useful. *The Java Language* [Arnold and Gosling, 1996] provides a good introduction to the Java language.

2.1 The dJVM as a State Machine

The dJVM state consists of a set of class declarations, a set of objects (instances of classes), a call stack, and a status flag.

To support multiple threads, the dJVM state would need to be extended to include multiple call-stacks with new call-stacks being created each time a new thread was started. Future Extension

---

[1] Of course, the description of ACL2 and the public libraries are also necessary for a full formal definition of the dJVM. Also, this report only describes the initial dJVM 0.5 model, so not all of the ultimate dJVM 1.0 has been modeled.

BNSDOCID: <XP 2161993A I >

The call stack contains stack frames representing each of the method invocations whose execution has not yet completed.

dJVM execution proceeds by taking the next step (i.e., executing the next instruction) for the top-most (most recently created) stack frame. If that executes a JVM instruction (other than one of the "return" instructions), then the state of that frame is updated (e.g., adjusting the local data stack, local variables, and local PC appropriately), and execution continues. If the instruction was one of the "return" instructions, then that stack frame is popped from the evaluation stack, and any returned-value is pushed onto the data stack of the new top-most stack frame. If an error occurred during execution of the step, then the dJVM 0.5 halts with status indicating what error occurred.

Future Extension

The dJVM 0.5 model does not support throwing or handling exceptions. When an exception would be thrown by the JVM, the dJVM halts.

The dJVM 0.5 should be extended to support throwing and catching exceptions.

2.2 Overview of dJVM State

The dJVM state includes four basic components:

- a *heap* of objects,
- a *call stack*, which records information about currently executing methods.
- a *class table* mapping class names to class definitions, and
- a *status* flag, indicating whether the dJVM is halted.

The *call stack* contains *call frames*. Each call frame corresponds to a method invocation that has not yet completed.

Future Extension

The dJVM 0.5 state can be extended to support multi-thread computations simply by adding multiple call stacks.

A call frame contains:

- an operand stack
- a vector of local variables (called "registers" in the early JVM specification)
- a method

⋮

NSDOCID: <XP 2181893A I >

2.3 Instructions as State Transitions

2.4 An Executable Interpreter

2.5 Naming conventions

Instruction definitions are usually named in the style of `djvm-execute-XXX` with *XXX* replaced with the instruction name.

2.5.1 Standard Pieces of Instruction Definitions

Instruction definitions are broken into several parts. For each instruction we will (normally) define one function to correspond to each of these parts. By convention, each of these functions have standard names and take the standard argument lists:

1. `(xxx-wff-inst? inst)`

   This function tests whether its argument is a well-formed dJVM instruction, where *xxx* stands for the name of the instruction whose form is to be checked.

2. `(xxx-proper-arg-types? inst frame)`

   This function tests whether the *types* of the *instruction arguments* and *operands* in the frame are properly typed. Instruction arguments are the values that appear within an instruction. In addition to instruction arguments, most instructions also take additional operands from the operand stack or from local variables.

   Note that this test is made with respect to the current frame, but without respect to objects in the heap. This is meant to correspond to information that could be checked by a bytecode verifier at load time.

3. `(xxx-resolve-args inst djvm)`

   This function is a place-holder for potential resolution of instruction arguments or class initialization. In all of the current instruction definitions it is a no-op.

4. `(xxx-proper-arg-values? inst resolved-djvm)`

   This function tests whether the *values* of the instruction arguments and operands are appropriate. For example, this function checks that operands that reference objects in the heap identify instances of the appropriate reference types (or `null`). In contrast, the `xxx-proper-arg-types?` test could only check that the operand was *some* reference type, since resolving the actual reference type requires accessing the object in the heap.

   This function may accept values that are defined to trigger run-time exceptions. For example, it checks that the stack operands of the integer division instruction `idiv` are integers, but does not check that the denominator is non-zero. The `idiv` instruction is defined to signal a run-time exception if the denominator is zero.

   These tests may include tests normally considered part of class, field, and method resolution.

BNSDOCID: <XP 2161893A I >

5. `(djvm-execute-xxx-optimized instruction djvm)`

   This function defines the behavior of the *xxx* instruction assuming that all of the tests just mentioned succeeded. Thus, we know that the instruction is well-formed, that its arguments and operands all have proper types and values, and proper reference values (for operands that are of reference types).

6. `(xxx-proper-result-type-? inst new-djvm)`

   For instructions that leave a result on the operand stack, this function checks that the result is of an appropriate type. For example, `idiv-proper-result-type?` checks that after normal completion of the `idiv` instruction the top-most operand on the stack is of type `int`.

7. `(djvm-execute-xxx-inst inst djvm-with-pc-incremented)`

   This is the *defensive* definition of the instruction. After making the tests mentioned above it applies the *optimized* version, and checks the result type afterward.

Appendix C describes the forms and macros used for defining instructions in more detail.

Instruction Formats

Some instructions include small integer arguments. When an JVM instruction takes an argument that is an index into the constant pool, the corresponding dJVM instruction takes the corresponding value or values from the constant pool. For example, the JVM instruction `getfield` takes as its single argument an index identifying a `CONSTANT_Fieldref` in the constant pool. The dJVM instruction `getfield` takes three arguments, the three string constant items (class name, field name, and type signature) identified in the constant pool. Here is an example of the dJVM instruction:

```
(getfield "Point" "x" "I")
```

where "`Point`" is the name of a class of the instance to be accessed, "`x`" is the name of the field to be accessed, and "`I`" is the type signature of that field. As well as distributing the constant-pool entries into the instructions, the dJVM uses symbolic opcodes, rather than the numeric bytecode values. Both of these choices were made to make the dJVM states easier to examine and construct. They also help to simplify the dJVM model somewhat, allowing us to concentrate on the richer parts of the definition.

BNSDOCID <XP 2161893A I >

Chapter 4

Primitive Values

The dJVM manipulates data values. Within the dJVM state each data value has an associated type. A type defines the set of values that a variable may hold, that an input value or output value must be within.

The Java language defines seven primitive numeric types and allows declaration of `reference` types, whose values refer to class instances or `null`.[1]

We use the term *primitive values* to include both numeric values (i.e., values of the primitive numeric types) and reference values.

4.1 Primitive Types

The Java language has seven primitive types. They are defined in terms of the numeric values that variables of these types are permitted to hold.

1. `byte` values are in the range $[-2^7 .. 2^7 - 1]$
2. `short` values are in the range $[-2^{15} .. 2^{15} - 1]$
3. `int` values are in the range $[-2^{31} .. 2^{31} - 1]$
4. `long` values are in the range $[-2^{63} .. 2^{63} - 1]$
5. `char` values are in the range $[0 .. 2^{16} - 1]$
6. `float` (not used in the dJVM)
7. `double` (not used in the dJVM)

The JVM has direct support for four of these primitive types (`int`, `long`, `float`, and `double`), as well the `reference` type. The JVM does not directly support either `byte`, `char`, or `short` integer types. The "narrower" integer types (`byte`, `short`, and `char`) are not directly supported, but are represented as `int` values during processing.[2] The JVM operand

[1] The value `null` is defined in Java to be a distinct type, rather than a value shared by each reference type. At the level of the JVM the value `null` is a value that is shared by all reference types.

[2] The JVM provides support for storing `byte`, `short`, and `char` values in arrays. The dJVM 0.5 model does not support arrays.

BNSDOCID: <XP 2161893A I >

stack and local variables are defined to hold 32-bit values. So 64-bit `long` values that are stored on the operand stack or in local variables must be represented as two 32-bit entries.

The dJVM does not support floating point variables (i.e., types `float` or `double`).

Java also admits the `reference` type, which permits references to class instances.

4.2 Tagged Values

Within the dJVM operand values are stored as `tagged values`. Each of the primitive data values appearing in the dJVM state is actually a pair, consisting of a `type tag` and a `primitive value` of the appropriate type. Each primitive value is tagged with its type (`int`, `long`, or `ref`).

The JVM specifies that certain internal data is stored as *abstract words*. Each abstract word can at least hold values corresponding to 32-bit two's-complement integers (i.e., corresponding to the `int` primitive type). The operand stack and local variables, which will be components of call frames (section 8.4, page 82) are defined to hold abstract words. Because a `long` value will not fit into a single abstract word, `long` values must be stored as two entries on the operand stack or as two local variables.

Future Extension

This will affect atomicity of dJVM operations when the dJVM is extended to model concurrency, because access to multiple words may require multiple memory references, and so can be interleaved with other operations. Be warned!

The predicate `tv-p` tests whether its argument is a tagged value — that is, a pair whose first element is a type tag (e.g., such as `:int`), which satisfies the predicate `jvm-type?`, and whose second element is any atomic value (e.g., an integer) appropriate to the type tag.

We use the macro `def-tv` to define recognizers for several sorts of tagged-value terms. Each tagged value is a type tag and a value satisfying an appropriate predicate. The accessors `tv-tag` and `tv-val` extract the tag and value (respectively) from a tagged-value.

The form

```
(def-tv tv-ref-p (:ref unsigned-int-p))
```

defines the recognizer `tv-ref-p`. Subsequently the term `(tv-ref-p alpha)` has the meaning

```
(and (weak-tv-p alpha)
     (equal (tv-tag alpha) :ref)
     (unsigned-int-p (tv-val alpha)))
```

The test `(weak-tv-p alpha)` checks that we can safely apply `tv-tag` and `tv-val` to access the components of `alpha`.

NSDOCID <XP 2161893A I >

```
(defmacro def-tv (name car-and-cadr-p)
  (declare (xargs :guard (and (consp car-and-cadr-p)
                              (symbolp (car car-and-cadr-p))
                              (symbolp (cadr car-and-cadr-p)))))
  (let ((car-p (car car-and-cadr-p))
        (cadr-p (cadr car-and-cadr-p)))
    `(defmacro ,name (x)
       (list 'and
             (list 'weak-tv-p x)
             ,(if (keywordp car-p)
                  `(list 'equal (list 'tv-tag x) ,car-p)
                `(list ',car-p (list 'tv-tag x)))
             ,(if (keywordp cadr-p)
                  `(list 'equal (list 'tv-val x) ,cadr-p)
                `(list ',cadr-p (list 'tv-val x)))))))
```

Reference values are addresses of heap objects. Since they can reside in the operand stack, and values in the operand stack must be representable in abstract words, heap addresses are implicitly limited to 32-bits.

Note on JV Semantics

Below are recognizers for the dJVM tagged values. The tags `:long-top-half` and `:long--bot-half` identify the two parts of a `long` value when it is represented as a pair of abstract words. The tag `:long` is used when the long integer value is represented as a single tagged value.[3]

```
(def-tv tv-int-p (:int int-value-p))

(def-tv tv-long-p (:long long-value-p))

(def-tv tv-ref-p (:ref unsigned-int-value-p))

(def-tv tv-long-top-half-p (:long-top-half unsigned-int-value-p))

(def-tv tv-long-bot-half-p (:long-bot-half unsigned-int-value-p))
```

Appendix B of this document defines bounded integer arithmetic for `int` and `long` integer values. The predicate `int-value-p` recognizes integer values in the range permitted by the type `int`. Similarly the predicate `long-value-p` recognizes integer values permitted by the type `long`. The predicate `unsigned-int-value-p` recognizes integer values in the range $[0 .. 2^{32} - 1]$. The JVM instructions do not support unsigned arithmetic operations, but unsigned 32-bit integers are used in the dJVM to represent parts of long values values and heap addresses.

```
(defun value-of-type? (type-tag value)
  (case type-tag
    (:int    (INT-VALUE-P    value))
    (:long   (LONG-VALUE-P   value))
```

[3] Obviously such a tagged long integer value cannot reside on the operand stack, or in a local variable. But it will be able to reside in an instance field, as we will see in section 4.3, page 27.

BNSDOCID <XP 2161893A I >

```
319    (:ref      (UNSIGNED-INT-VALUE-P     value))
       ((:Long-Top-Half
         :Long-Bot-Half
319      )        (UNSIGNED-INT-VALUE-P     value))
       (otherwise nil)
       ))
```

The function make-tv constructs a tagged-value pair from a type tag and a value. It does not check that the value is appropriate for the given type tag. In the contexts in which we use make-tv, we know that the value is in fact appropriate, and ACL2'S guard verification will generally enforce this requirement.[4]

As mentioned above, the predicate weak-tv-p tests whether its argument has the *form* of a tagged value, suitable for accessing via tv-tag and tv-val. Concretely, it checks that its argument is a proper list of two elements.

```
(defun make-tv (tag val)
  (declare (xargs :guard t))
  (list tag val))

(defun weak-tv-p (x)
  (declare (xargs :guard t))
  (and (consp x)
       (consp (cdr x))
       (not (consp (cddr x)))))

(defun tv-tag (x)
above  (declare (xargs :guard (WEAK-TV-P x)))
  (first x))

(defun tv-val (x)
above  (declare (xargs :guard (WEAK-TV-P x)))
  (second x))


(defthm weak-tv-p-make-tv
above  (WEAK-TV-P (make-tv tag val)))

(defthm make-tv-pieces
above  (and (equal (tv-tag (MAKE-TV x y)) x)
above       (equal (tv-val (MAKE-TV x y)) y))
  :hints (("Goal" :in-theory (enable make-tv tv-tag tv-val))))
```

[4] Tagged values will ultimately be stored on the operand stack, in local variables, or in an instance field. All of these are required to be properly-tagged values, and guard verification will enforce this requirement.

NSDOCID <XP_ 2161893A_ I >

4.3 Stack Values and Field Values

We distinguish between values that are permitted on the operand stack (or in local variables) and values that may appear in instance and class fields. The distinction concerns the handling of `long` integer values. The operand stack always manipulates 32-bit data.[5] So when a 64-bit integer (`long`) value must be pushed onto the stack, it is divided in half, and the two halves are pushed onto the stack individually. However, we do not make the same restriction on `long` values stored in an object field. A field declared to hold a `long` value can hold the entire 64-bit integer value.

The Java Virtual Machine Specification describes stack values as:

> No mention has been made of the storage requirements for values of various Java Virtual Machine types, only ranges those values may take. ...The Java Virtual Machine defines an abstract notion of a *word* that has a platform-specific size. A word is large enough to hold a value of type `byte`, `char`, `short`, `int`, `float`, reference, `returnAddress`, or to hold a native pointer. Two words are large enough to hold values of the larger types, `long` and `double`. Java's run-time data areas are all defined in terms of these abstract words.
> [Lindholm and Yellin, 1996, §3.4, p. 61]

Our discussion in this section is slightly more concrete than that, in that we specify that stack values must be representable with words of only 32 bits. This addition is intended to make the dJVM model more closely resemble current-day JVM implementations. Further, during execution of the dJVM model, we divide `long` values precisely into two 32-bit halves when storing the long value on the operand stack or into local variables. Again this is intended to make execution states more closely resemble common implementations.

We should first define the specifications and functions axiomatically, and then provide executable versions. This will clarify the formal requirements *from the* executable implementation *of the dJVM.* — Author's Note

We refer to values that are permitted on the operand stack as *stack values*, and we refer to values that are permitted in object fields as *field values*. Observe that these two sets are not disjoint. Tagged values of type `:ref` and `:int` are both legal stack values and field values.

When data moves between object fields and the operand stack, `long` values will have to be converted between their 64-bit integer representation and the corresponding two 32-bit "half-long" values. We will see this conversion as part of the `getfield`, `putfield`, `getstatic`, and `putstatic` instructions

> Implementors are free to decide the appropriate way to divide a 64-bit data value between two local variables.
> [Lindholm and Yellin, 1996, §3.6.1, p. 67]
>
> Implementors are free to decide the appropriate way to divide a 64-bit data value between two operand stack words.
> [Lindholm and Yellin, 1996, §3.6.2, p. 67]

[5] Of course in the dJVM the 32-bit data values are always tagged.

BNSDOCID <XP 2161893A I >

Author's Note

Since these descriptions appear in two separate sections, presumably the implementor is free to choose differently for the stack and for local variables. This would significantly complicate the instructions for dealing with loading and storing double-word values, and this possibility is not addressed in those instructions.

Author's Note

*Does the JLS or JVMS say anything about how values are stored in **object fields**?*

I think the implementor is free to do what he likes in the object representation.

However they do state that accessing `long` and `double` fields is not an atomic operation. Since `long` and `double` are stored as two words in the stack, we know that such accesses will at least make two references to the local frame, and this may be enough to imply that such accesses may not be atomic — regardless of how the `long` or `double` field values are actually stored.

They explicitly state that `byte`, `short`, and `char` values are stored as "word" values on the stack and in local variables (except for arrays). They define a notion of an abstract word of unspecified size, but capable of representing all `int` values. The JVM always reserves two words for long and double values, even though on some implementations (e.g., platforms with 64-bit words) a single word might be capable of representing `long` and `double` values.

The JVM specification does not state how `long` values should be represented in fields within an instance. We have chosen to represent `long` values as single entries in an instance. Thus, an instance field may contain a value of any type except the `long-top-half` and `long-bot-half` types, which are only permitted on the operand stack and in local variables. Collectively we refer to the values permitted in instance fields as *field values*. The predicate `fv-p` recognizes legal field values.

```
(defun fv-p (x)
  (declare (xargs :guard t))
  (or (TV-INT-P  x)        25
      (TV-LONG-P x)        25
      (TV-REF-P  x)))      25
```

Local variables and the operand stack may hold values of any type, except `long`. `Long` values must be represented using two smaller values, one of type `long-top-half` and one of type `long-bot-half`. Collectively we refer to these values as legal `stack values`. The predicate `sv-p` recognizes legal stack values.

```
(defun sv-p (x)
  (declare (xargs :guard t))
  (or (TV-INT-P x)                 25
      (TV-REF-P x)                 25
      (TV-LONG-TOP-HALF-P x)       25
      (TV-LONG-BOT-HALF-P x)))     25
```


NSDOCID <XP 2161893A I >

above

```
(defthm sv-p-is-weak-tv-p-forward
  (implies (SV-P x)
           (WEAK-TV-P x))
  :hints (("Goal" :in-theory (enable sv-p)))
  :rule-classes (:forward-chaining))

(defthm sv-p-if-fv-p-and-not-long
  (implies (and (FV-P x)
                (not (TV-LONG-P x)))
           (SV-P x))
  :hints (("Goal" :in-theory (enable fv-p sv-p))))

(defthm sv-p-recomposed-sv
  (and (implies (SV-P x)
                (sv-p (MAKE-TV (tv-tag x) (tv-val x))))
       (implies (and (SV-P tv)
                     (equal tag (TV-TAG tv))
                     (equal val (TV-VAL tv)))
                (sv-p (MAKE-TV tag val)))))

(defthm sv-p-make-tv-int
  (implies (INT-VALUE-P val)
           (sv-p (MAKE-TV :int val))))

(defthm sv-p-remake-tv-ref-p
  (implies (TV-REF-P x)
           (SV-P (MAKE-TV :REF (tv-val x)))))

(defthm sv-p-make-tv-unsigned-int-case
  (implies (and (UNSIGNED-INT-VALUE-P val)
                (member tag '(:ref :long-top-half :long-bot-half)))
           (sv-p (MAKE-TV tag val))))

#+NOT-NEEDED
(defthm sv-p-make-tv-ref
  (implies (UNSIGNED-INT-VALUE-P x)
           (SV-P (MAKE-TV :REF x)))
  :hints (("Goal" :in-theory (enable sv-p))))

(defthm sv-p-make-tv-long-xxx-half
  (implies (integerp x)
           (and (sv-p (make-tv :long-bot-half (LONG-BOT-HALF x)))
                (sv-p (make-tv :long-top-half (LONG-TOP-HALF x)))))
  :hints (("Goal" :in-theory (enable tv-tag
                                     tv-val
                                     sv-p
                                     ))))
```

Define predicates that recognize lists of stack values (`sv-listp`) and lists of field values (`fv-listp`).

```
(deflist sv-listp (l)
```


BNSDOCID <XP 2161893A I >

```
  sv-p)


(defthm sv-listp-is-true-listp-forward
29  (implies (SV-LISTP x)
             (true-listp x))
  :rule-classes (:forward-chaining))

(in-theory (disable sv-listp-true-listp))


(deflist fv-listp (1)
  fv-p)


(defthm fv-listp-is-true-listp-forward
above  (implies (FV-LISTP x)
             (true-listp x))
  :rule-classes (:forward-chaining))

(in-theory (disable fv-listp-true-listp))

(defthm sv-p-defn
25  (implies (or (TV-REF-P x)
25               (TV-INT-P x)
25               (TV-LONG-TOP-HALF-P x)
25               (TV-LONG-BOT-HALF-P x))
28           (SV-P x)))

(defthm fv-p-defn
25  (implies (or (TV-REF-P x)
25               (TV-INT-P x)
25               (TV-LONG-P x))
28           (FV-P x)))
```

The function `tv-ref-listp` recognizes a list of (tagged) object references.

```
(deflist tv-ref-listp (1)
  tv-ref-p)
```

Below are type prescription lemmas giving type information for applications of `tv-val` to tagged values. Obviously the type yielded by `tv-val` depends on the type-tag of the tagged value.

First we observe that all of the standard tagged-values have integers as their `tv-val` component. Then we give the type information as type-prescription lemmas and provide tighter integer-bounds in rewrite rules.

```
(defthm tv-val-type-prescription
28  (implies (or (SV-P x)
```


BNSDOCID: <XP 2161893A I >

28
26

```
            (FV-P x))
        (integerp (TV-VAL x)))
  :rule-classes (:rewrite :type-prescription))

(defthm naturalp-tv-val-type-prescription
  (implies (or (TV-REF-P x)
               (TV-LONG-TOP-HALF-P x)
               (TV-LONG-BOT-HALF-P x))
           (naturalp (TV-VAL x))))

(defthm int-value-p-tv-int-p
  (implies (TV-INT-P x)
           (INT-VALUE-P (tv-val x))))

(defthm long-value-p-tv-long-p
  (implies (TV-LONG-P x)
           (LONG-VALUE-P (tv-val x))))

(defthm unsigned-int-value-p-tv-ref-p
  (implies (TV-REF-P x)
           (UNSIGNED-INT-VALUE-P (tv-val x))))

(defthm unsigned-int-value-p-tv-long-top-half-p
  (implies (TV-LONG-TOP-HALF-P x)
           (UNSIGNED-INT-VALUE-P (tv-val x))))

(defthm unsigned-int-value-p-tv-long-bot-half-p
  (implies (TV-LONG-BOT-HALF-P x)
           (UNSIGNED-INT-VALUE-P (tv-val x))))

(defun tv-one-word-p (x)
  (declare (xargs :guard t))
  (or (TV-INT-P x)
      (TV-REF-P x)))

(defun tv-two-word-p (x)
  (declare (xargs :guard t))
  (or (TV-LONG-TOP-HALF-P x)
      (TV-LONG-BOT-HALF-P x)))


(in-theory (disable make-tv tv-tag tv-val))
```

4.4 The Null object

We should also define functions that give the default value for each primitive type - including (ref-to-null) for reference types.

Note: there is no default initial value for `:addr` *values. We only need default values for declared fields, and fields cannot be declared of type* `address`.

Author's Note

BNSDOCID <XP 2161993A I >

```
(defun addr-of-null ()
  0)

(defun ref-to-null ()
  (make-tv :ref (ADDR-OF-NULL)))

(defun ref-to-null-p (x)
  (equal x (REF-TO-NULL)))
```

141

above

4.5 64-bit wide types

`Wide-type-tags` lists the type tags associated with 64-bit data values. Wide values can only be stored on the stack or in local variables if they are split into two parts that will fit within the 32-bit size of individual stack entries and local variables. For each type tag denoting a wide (64-bit) value, we define two associated type tags that denote two 32-bit values to be associated with the 64-bit value. We call these two 32-bit values the "top half" and the "bottom half" of the wide value.

We define functions for manipulating 64-bit wide values, for splitting them into two separate 32-bit values (their top and bottom "halves") and for combining the two halves back into the single wide-value.

Future Extension

The dJVM 0.5 model only supports 64-bit wide integers (i.e., the `long` data type). A future extension could support additional 64-bit types (e.g., `double`.) We would then define a general notion of wide values, *that would include all 64-bit field values.*

```
(defun tv-top-half-p (x)
  (declare (xargs :guard t))
  (TV-LONG-TOP-HALF-P x))

(defun tv-bot-half-p (x)
  (declare (xargs :guard t))
  (TV-LONG-BOT-HALF-P x))

(defun tv-wide-p (x)
  (TV-LONG-P x))

(defun parts-of-same-wide-type (top-half bot-half)
  (declare (xargs :guard (and (sv-p top-half)
                              (sv-p bot-half))))
  (case (tv-tag top-half)
    (:long-top-half (equal (tv-tag bot-half) ':long-bot-half))
    ))
```

25

25

25

NSDOCID <XP 2181993A I >

```
(defun tv-long-top-half (x)
  (declare (xargs :guard (TV-LONG-P x)))
  (make-tv ':long-top-half (LONG-TOP-HALF (tv-val x))))

(defun tv-long-bot-half (x)
  (declare (xargs :guard (TV-LONG-P x)))
  (make-tv ':long-bot-half (LONG-BOT-HALF (tv-val x))))


(defthm sv-p-tv-long-top-half
  (sv-p (TV-LONG-TOP-HALF x))
  :hints (("Goal" :in-theory (enable  sv-p
                                      tv-long-top-half
                                      value-of-type?))))

(defthm sv-p-tv-long-bot-half
  (sv-p (TV-LONG-BOT-HALF x))
  :hints (("Goal" :in-theory (enable  sv-p
                                      tv-long-bot-half
                                      value-of-type?))))


(defun tv-wide-top-half (x)
  (declare (xargs :guard (and (FV-P x)
                              (TV-WIDE-P x))))
  (case (TV-TAG x)
    (:long (TV-LONG-TOP-HALF x))))

(defun tv-wide-bot-half (x)
  (declare (xargs :guard (and (FV-P x)
                              (TV-WIDE-P x))))
  (case (TV-TAG x)
    (:long (TV-LONG-BOT-HALF x))
    ))

(defthm tv-wide-half-type-prescription
  (implies (TV-WIDE-P x)
           (and (sv-p (TV-WIDE-TOP-HALF x))
                (sv-p (TV-WIDE-BOT-HALF x)))))

(defun make-tv-wide-value (top-half bot-half)
  (declare (xargs :guard (and (TV-TOP-HALF-P top-half)
                              (TV-BOT-HALF-P bot-half)
                              (PARTS-OF-SAME-WIDE-TYPE top-half bot-half))))
  (case (tv-tag top-half)
    (:long-top-half (MAKE-TV ':long
                             (MAKE-LONG (tv-val top-half) (tv-val bot-half))))
    ))


(defthm make-tv-wide-value-type-prescription
  (implies (and (TV-TOP-HALF-P x)
                (TV-BOT-HALF-P y)
                (PARTS-OF-SAME-WIDE-TYPE x y))
           (tv-wide-p (MAKE-TV-WIDE-VALUE x y))))
```


BNSDOCID <XP _ 2161893A I >

Below are a few general recognizers for properly-typed tagged-values, for lists of tagged-values, and for lists of reference values.

```
     (defun tv-wide-value-p (x)
25     (TV-LONG-P x))

     (defthm tv-long-half-facts
25     (implies (TV-LONG-P x)
25              (and (TV-LONG-TOP-HALF-P         (tv-long-top-half x))
33                   (tv-top-half-p              (TV-LONG-TOP-HALF x))
33                   (sv-p                 (TV-LONG-TOP-HALF x))
33                   (equal                (tv-tag (TV-LONG-TOP-HALF x)) ':long-top-half)
33                   (sv-p                       (TV-LONG-TOP-HALF x))
319                  (UNSIGNED-INT-VALUE-P (tv-val (tv-long-top-half x)))
25                   (TV-LONG-BOT-HALF-P         (tv-long-bot-half x))
33                   (tv-bot-half-p              (TV-LONG-BOT-HALF x))
33                   (sv-p                 (TV-LONG-BOT-HALF x))
33                   (equal                 (tv-tag (TV-LONG-BOT-HALF x)) ':long-bot-half)
33                   (sv-p                       (TV-LONG-BOT-HALF x))
319                  (UNSIGNED-INT-VALUE-P (tv-val (tv-long-bot-half x)))))
     :hints (("Goal" :in-theory (enable make-tv
                                        sv-p
                                        tv-tag
                                        tv-val
                                        ))))

     (in-theory (disable tv-long-top-half tv-long-bot-half))

     (defthm weak-tv-p-tv-long-top-half
33     (weak-tv-p (TV-LONG-TOP-HALF x))
       :hints (("Goal" :in-theory (enable tv-long-top-half))))

     (defthm weak-tv-p-tv-long-bot-half
33     (weak-tv-p (TV-LONG-BOT-HALF x))
       :hints (("Goal" :in-theory (enable tv-long-bot-half))))

     (defthm wide-half-facts
28     (implies (and (sv-p x)
above                (TV-WIDE-VALUE-P x))
33              (and (sv-p (TV-WIDE-TOP-HALF x))
33                   (sv-p (TV-WIDE-BOT-HALF x)))))

     (defun jvm-type-tags ()
       '(:int :long :long-top-half :long-bot-half :ref))

     (defun jvm-type-tag? (x)
above  (memberp x (JVM-TYPE-TAGS)))
```


NSDOCID <XP 2161893A I >

28
```
(defthm jvm-type-tag-if-tv-p
  (implies (or (SV-P x)
               (FV-P x))
           (JVM-TYPE-TAG? (tv-tag x))))

(deflist weak-tv-listp (1)
  weak-tv-p)


(defthm weak-tv-listp-is-true-listp-forward
  (implies (WEAK-TV-LISTP x)
           (true-listp x))
  :rule-classes (:forward-chaining))

(in-theory (disable weak-tv-listp-true-listp))

(defthm naturalp-listp-is-true-listp-forward
  (implies (NATURALP-LISTP x)
           (true-listp x))
  :hints (("Goal" :in-theory (enable naturalp-listp)))
  :rule-classes :forward-chaining)
```

We often have to reason about values satisfying `tv-p` and `weak-tv-p`. To avoid additional clutter in our proofs, we disable those definitions. So, we must define rules for how those concepts should relate in proofs. Any value that satisfies `tv-p` must also satisfy `weak-tv-p`. So we define a forward-chaining rule that says "whenever a value is known to satisfy `tv-p`, assert that it also satisfies `weak-tv-p`." We state a similar rule relating `weak-tv-listp` to `tv-listp`.

```
(defthm weak-tv-p-if-sv-p-forward
  (implies (SV-P x)
           (WEAK-TV-P x))
  :rule-classes (:forward-chaining))

(defthm weak-tv-p-if-fv-p-forward
  (implies (FV-P x)
           (WEAK-TV-P x))
  :rule-classes (:forward-chaining))

(deflist weak-tv-listp (1)
  weak-tv-p)

(defthm weak-tv-listp-if-sv-list-p-forward
  (implies (SV-LISTP x)
           (WEAK-TV-LISTP x))
  :rule-classes (:forward-chaining))

(defthm weak-tv-listp-if-fv-list-p-forward
  (implies (FV-LISTP x)
           (WEAK-TV-LISTP x))
  :rule-classes (:forward-chaining))
```


BNSDOCID <XP 2161993A I >

4.5.1 Extended Type Tags

So far we have used tags to mark data values used in the stack and in object fields. Now we extend the notion of tags to include *abstract tags* that allow specification of larger sets of data values. These extended tags will be used to describe constraints on part of the dJVM state. For example, the extended tag `:one-word-type` will be used to indicate that a tagged value must be of a one-word data type (i.e., anything but a two-word data type). These extended type tags will be used as part of the pattern language we use to define the "simple" dJVM instructions later on.

```
(deflist jvm-type-tag-listp (1)
  jvm-type-tag?)
```

```
(in-theory (disable jvm-type-tag-listp-true-listp))
```

```
(defun extended-type-tags ()
  '(:same
    :one-word-type
    :not-top-half
    :not-bot-half
    ))
```

34 above

```
(defun extended-type-tag? (x)
  (or (JVM-TYPE-TAG? x)
      (memberp x (EXTENDED-TYPE-TAGS))))
```

```
(deflist extended-type-tag-listp (1)
  extended-type-tag?)
```

The tags used in primitive tagged values can be used as type specifications. The use of a particular type in a type specification indicates that the corresponding tagged value should be of the indicated type. This is sufficient to discuss the primitive types, but is not adequate to specify reference types — that is, values that refer to instances in the heap. The function `extended-jvm-type?` extends the standard type tags to allow the type-tag `:ref` to be paired with a string. (In our use later, the string will be required to be a class name, but for the moment we only impose the syntactic requirement.)

26 26 26 34

```
(defun extended-jvm-type? (x)
  (if (WEAK-TV-P x)
      (and (equal (TV-TAG x) ':ref)
           (stringp (TV-VAL x)))
    (JVM-TYPE-TAG? x)))
```

```
(deflist extended-jvm-type-listp (1)
  extended-jvm-type?)
```


NSDOCID <XP 2161893A | >

```
(in-theory (disable extended-jvm-type-listp-true-listp))
```

```
(in-theory (disable extended-type-tag-listp-true-listp))

(in-theory (disable value-of-type?))
```

Well-formed operand stacks. A well-formed operand stack is a sequence of stack values with the added constraint that every long-top-half value must immediately precede a long-bot-half value, and every long-bot-half value must immediately follow a long-top-half value. All well-formed JVM states will satisfy this predicate. Since the dJVM relies on run-time tests to validate the well-formedness of the JVM state, we will not rely on states being well formed. Each operation that alters the operand stack will check that it is not given an ill-formed stack and that its stack manipulations do not leave an ill-formed stack.

Future Extension

This is the sort of global state-invariant we expect the JVM to preserve during execution. Such properties are proven inductively; if an initial state satisfies the property and every step (executing a single instruction) preserves the property, then the property is invariant during execution.

The dJVM 0.5 model has not been proven to preserve this global invariant. After such a proof is done, some of the run-time defensive checks could be elided.

```
(defun valid-stack? (stk)
  (declare (xargs :guard (SV-LISTP stk)
                  :guard-hints (("Goal" :in-theory (disable weak-tv-p)))))
  (if (endp stk)
      t
    (if (TV-ONE-WORD-P (car stk))
        (VALID-STACK? (cdr stk))
      ;; It's part of a two-word type...
      (and (TV-LONG-TOP-HALF-P (car stk))
           (consp (cdr stk))
           (TV-LONG-BOT-HALF-P (cadr stk))
           (VALID-STACK? (cddr stk))))))
```


BNSDOCID: <XP ... 2161893A _I >

NSDOCID: <XP __2161893A _ I >

```
(defun method-type-sig-p-internal (sig i)
  "Parse sequential field-types in a signature
   until we hit the end of the string or a right parenthesis."
  (declare (xargs :guard (and (stringp sig)
                              (naturalp i)
                              (<= i (length sig)))
                  :measure (max 0 (- (length sig) (acl2-count i)))
                  :guard-hints (("Goal" :in-theory (disable length)))))
  (if (and (naturalp i)
           (< i (length sig)))
      (if (equal (char sig i) #\))
          (RETURN-TYPE-SIG-P sig (1+ i))
        (let ((next-field-index (FIELD-TYPE-SIG-INTERNAL-P sig i)))
          (and (not (null next-field-index))
               (METHOD-TYPE-SIG-P-INTERNAL sig next-field-index))))
    nil))

(defun method-type-sig-p (sig)
  "Recognize a method type-signature."
  (declare (xargs :guard t))
  (and (stringp sig)
       (> (length sig) 0)
       (equal (char sig 0) #\()
       (METHOD-TYPE-SIG-P-INTERNAL sig 1)))

(defthm stringp-if-method-type-sig-p
  (implies (METHOD-TYPE-SIG-P x)
           (stringp x))
  :rule-classes (:forward-chaining))
```

7.5 Type-Tag Lists

Now we define a translator from a type-signature string to a list of type tags, which is more easily used for run-time type-checking.

Note that the type-signature of a single field may translate into a list of one or two type tags, depending on whether the type-signature represents a one-word type (e.g., `int`) or a two-word type (e.g., `long`).

Note also that `type-list-for-field-type-sig` does not return a list of symbols. Reference types are represented as the cons-pair `:ref` and the class name (as a string). We define the predicate `extended-jvm-type-listp`, which is similar to the predicate `jvm-type-listp`, to recognize these extended type-lists.

```
(defun type-list-for-field-type-sig-internal (sig i)
  (declare (xargs :guard (and (stringp sig)
                              (naturalp i)
                              (<= i (length sig)))))
  (if (< i (length sig))
      (case (char sig i)
        (#\I '(:int))
```


BNSDOCID: <XP_ 2161893A I >

```
        (#\J '(:long-top-half :long-bot-half))
58      (#\L (list (list ':ref (FIELD-TYPE-SIG-CLASS-NAME sig (1+ i)))))
        (otherwise '(:error)))
    '(:error)))
```

The function `type-list-for-field-type-sig` translates a field type-signature into a list of type tags as described above. The tag list is used for run-time type-checking when accessing instance fields and when invoking or return from methods.

```
(defun type-list-for-field-type-sig (sig)
  "Translate a field type-signature to a list of extended-type tags."
  (declare (xargs :guard (and (stringp sig)
60                             (FIELD-TYPE-SIG-P sig))))
61 (TYPE-LIST-FOR-FIELD-TYPE-SIG-INTERNAL sig 0))

(defun type-list-for-method-parameters-internal (sig i)
  (declare (xargs :guard (and (stringp sig)
                              (naturalp i)
                              (<= i (length sig)))
                  :measure (max 0 (- (length sig) (acl2-count i)))
                  ))
  (if (or (not (naturalp i))
          ;;(not (stringp sig))
          (>= i (length sig)))
      nil
59   (let ((next-field-index (FIELD-TYPE-SIG-INTERNAL-P sig i)))
       (if (not (null next-field-index))
61         (append (TYPE-LIST-FOR-FIELD-TYPE-SIG-INTERNAL sig i)
above              (TYPE-LIST-FOR-METHOD-PARAMETERS-INTERNAL sig next-field-index))
         (if (equal (char sig i) #\))
             nil
           '(:error))))))
```

The function `type-list-for-method-parameters` returns a list of extended-type tags for a list of stack values to be a well-typed parameter list satisfying the given type-signature.

```
(defun type-list-for-method-parameters (sig)
61 (declare (xargs :guard (and (stringp sig)
                              (METHOD-TYPE-SIG-P sig))))
  (if (and (> (length sig) 0)
           (equal (char sig 0) #\())
above  (TYPE-LIST-FOR-METHOD-PARAMETERS-INTERNAL sig 1)
    '(:error)))

(in-theory (disable field-type-sig-p
                    method-type-sig-p
                    type-list-for-method-parameters
                    type-list-for-field-type-sig))
```


'NSDOCID <XP 2151993A I >

Chapter 9

Manipulating the dJVM State

This chapter defines the internal operations that alter the dJVM state. The dJVM instructions will be defined in terms of these operations.

9.1 Checking Stack Signatures

Most Java instructions require that the top few elements of the stack contain values of specified data types. For example, the integer addition operation (`iadd`) requires that the top two elements of the stack contain integer values. The predicate `stk-sig-top?` is used to state just such requirements. The precondition for `iadd` would be stated as:

```
(stk-sig-top? stk-sig :int :int).
```

The predicate `stk-sig-top?` is defined as a macro, so that it can take a variable number of arguments. Remark on ACL2

Note on reference types: Although the an extended-type tag list, such as returned by the function `type-list-for-field-type-sig-internal`, includes both `:ref` and the specified class name for a reference type, the function `stk-sig-top?` only checks that the stack value is a reference value. `stk-sig-top?` is intended to check the basic type-safety preconditions for JVM instructions.

While some JVM instructions require an argument to be a reference value, some of them require (as a precondition) that value be a reference to an instance of particular class. For example, the `athrow` instruction has a precondition that its argument be a reference type and that the referenced object is an instance of class `java.lang.Throwable`. The bytecode

BNSDOCID: <XP 2161893A_ I >

verifier checks both of these preconditions. The bytecode verifier attempts to resolve the the declared class of the object being thrown. If resolution fails, the bytecode verification fails. In contrast, some other instructions, such as `checkcast` and `instanceof`, take *class names* as arguments; these instruction argument values are represented as strings[1] and they are not resolved to a class object until execution time.

We provide a bottom-up description of these macros.

Author's Note

I don't believe that `unquote-list-elements` is needed any longer. Initially I thought this would be used in top-level code, and hadn't decided whether type labels should be quoted or not in that context.

However, currently this is only used in the `define-djvm-operation` macro. And those arguments are not explicitly quoted. So, I should pull this out and recompile and reverify.

```
(defun unquote-list-elements (list)
  (declare (xargs :guard (true-listp list)))
  (if (endp list)
      nil
    (cons (if (and (true-listp (car list))
                   (equal (caar list) 'quote))
              (cadar list)
            (car list))
          (UNQUOTE-LIST-ELEMENTS (cdr list)))))
```

above

```
(defun stk-sig-top-type-listp (x)
  (declare (xargs :guard (true-listp x)))
  (if (endp x)
      t
    (and (or (symbolp (car x))
             (and (consp (car x))
                  (eql (car (car x)) ':ref)))
         (STK-SIG-TOP-TYPE-LISTP (cdr x)))))
```

above

In definition of `assert-value-typed-properly` below, the value of the parameter `variable` is expected to be an ACL2 expression that accesses a typed-value within the dJVM state (e.g., an element of the operand stack, or a local variable). The macro manipulates ACL2 forms, composing new forms from old ones.

We use an extended notion of type specifier here, so that we can assert more abstract specifications than merely that a value is of a particular type (e.g., `:int`). The extended specifiers are:

`:one-word-type` accepts any primitive type whose values occupy one word (i.e., are not half of a two-word value).

[1] In the JVM the strings reside in the constant pool for the class. In the dJVM the string values are part of the instruction.

SDOCID: <XP__ 2161893A_I_>

Chapter 10

The Standard Form of Instructions

10.1 Java's Shared-Memory Model

---

dJVM 0.5 does not include threads and synchronization. So it does not need to model shared-memory. However, we mention some of the issues related to shared-memory which may be dealt with in a future extension of the dJVM.

Future Extension

The Java language includes threads and synchronization. In support of them the language includes the concepts of *locks* and *monitors*. It also includes an explicit model of memory access that addresses how each thread in a multithreaded program accesses a main memory shared by all threads.

The language definition specifies that there is a *main memory* that is shared by all threads, and that each thread has its own *working memory*. The main memory contains the "master copy" of every variable. It also contains a *lock* associated with each object in main memory. Each thread operates on values from its working memory, copying values between its working memory and main memory as necessary. Chapter 17 of the *Java Language Specification* addresses when such copying is necessary, as well as which memory operations are atomic, and ordering requirements between program actions (e.g., the level of reordering of actions within a single thread and the interleaving of actions between multiple threads).

10.2 Standard Parts of Instruction Definitions

Since we are defining the *defensive* Java Virtual Machine, each instruction must perform various tests to assure that its arguments and results are well-formed and type-correct. For each instruction we define a function that implements that instruction. That function performs all the required tests and (if the tests succeed) updates the state of the dJVM to reflect execution of the instruction. Since each instruction-implementation function requires these tests and halts with similar errors if any test fails, the body of each follows the same form.

Each instruction implementation consists of six steps:

BNSDOCID: <XP 2161893A I >

1. Check that the instruction is syntactically well-formed.

   (We could include checks that static local variable indices are less than the number of local varibles specified in the method declaration. But that would require the additional context of the method declaration, which we will skip here. This index range-checking is done in step 4 below.)

2. Check that the arguments are of the proper number and type.

   For objects, this merely checks that the operand is a reference type. It does not check the actual class of the operand.

3. Resolve any referenced classes, fields, or methods if necessary.

   (This step would perform dynamic class-loading if it were supported.)

4. Check that the argument values are permissible.

   This tests any required semantic preconditions for the instruction. For example, the `idiv` instruction must check that the divisor is non-zero, and the `getfield` instruction that the object reference is not null.

5. Execute the instruction and update the `pc` register.

6. Check that any result values are permissible.

   For example, method invocations and field accesses should leave a value of the required type on the operand stack. This test is degenerate for instructions that always yield a fixed-type result, but is required for "generic" instructions (e.g., `getfield`, `new`, *etc.*)

   These tests should always be redundant in a type-safe program. Since the dJVM is intended to detect type errors, these checks are made explicitly. However, these tests should be provably true if the loaded JVM methods are all type-safe.

7. Check that the operand stack hasn't overflowed.

8. Check that the updated `pc` register is valid if the current method is a bytecoded method.[1]

9. Check that if the call stack is empty, then the dJVM status is `:halted`.

   Strictly speaking this is not a requirement of the JVM definition. In the dJVM the `return` is defined to set the machine status to `:halted` if it is exiting from the only frame on the call stack. No other instruction that completes normally should result in an empty call stack. The other return instructions (`ireturn`, `areturn`, *etc.*) return a *result value* to a calling frame, and so they should never be executed unless there are at least two frames on the call stack. An unhandled error or exception could also halt the computation, but the dJVM does not yet handle exceptions.

NSDOCID <XP 2181893A I >

Author's Note

Where do exceptions fit in??? Any instruction can throw an exception.

Author's Note

The JVMS claims to document and order *all exceptions that can arise as a result of constant pool resolution. So we must be careful that we honor that order!*

> A Java Virtual Machine throws an object that is an instance of a subclass of the class `VirtualMachineError` when an internal error or resource limitation prevents it from implementing the *semantics of the Java Language*. [Emphasis added.] The Java Virtual Machine specification cannot predicate where resource limitations or internal errors may be encountered and does not mandate precisely when they can be reported. Thus, any of the virtual machine errors listed as subclasses of `VirtualMachineError` ... may be thrown at any time during the operation of the Java Virtual Machine.
> [JVMS §6.3, p. 152]

Author's Note

To some extent the JVMS description seems to intermingle the idea that the JVM is tied to Java semantics and that the JVM is a separately specified machine. The quote above is just one example.

For each instruction we define a separate function to accomplish each of these steps. By convention each of these functions takes a standard argument list as follows:

1. `(getfield-wff-inst?  inst)`
2. `(getfield-proper-arg-types?  inst frame)`
   This predicate takes the instruction and the current frame. It can check the type of values in local variable and on the operand stack, but it cannot examine objects in the heap.
3. `(getfield-resolve-args inst djvm)`
4. `(getfield-proper-arg-values?  inst resolved-djvm)`
   This predicate takes the instruction and the full dJVM state (after any class resolution, which in an extended dJVM might trigger dynamic class loading). Thus this predicate can examine heap objects and check their class.

[1] The JVM does not define the value of the `pc` register when the current method is a native method. So it only makes sense to ask whether the `pc` is valid when executing a bytecoded method.

The dJVM 0.5 does not currently support execution of native methods. However, to permit future support for native methods the dJVM 0.5 defines a representation of native methods in stack frames. Thus, for example, after execution of a `return` instruction the current method may be either a bytecoded method or a native method. Of course, if it's a native method then the dJVM 0.5 will subsequently halt, since it cannot currently execute native methods — but the `return` instruction will have completed normally.

BNSDOCID <XP__2161893A_ I >

This Page Blank (uspto)

XP 000679974

p. 63-64 = (2)

p.d 01-1997

Java uses a sophisticated class-checking mechanism to ward off breaches in security. By Gary McGraw and Edward Felten

Java Security and Type Safety

Java's ability to download, integrate, and execute code from a remote computer is a double-edged sword. On the positive side, the use of Java enables a computer to obtain new capabilities with little user intervention. In addition, Java requires no installation of hard-to-track-down and dubiously secure plug-in files. On the negative side, however, Java's intricate machinations leave a computer vulnernable to attack. A hostile Java applet could stealthily tamper with a host system's files or siphon off private data without the user's being aware of the damage until it's too late.

Java's designers did their best to make such malicious activities impossible by implementing a security model. This security model performs a number of checks before allowing a downloaded applet to execute. (For additional information on Java security, see "Plugs for Java's Security Holes" on page 76.)

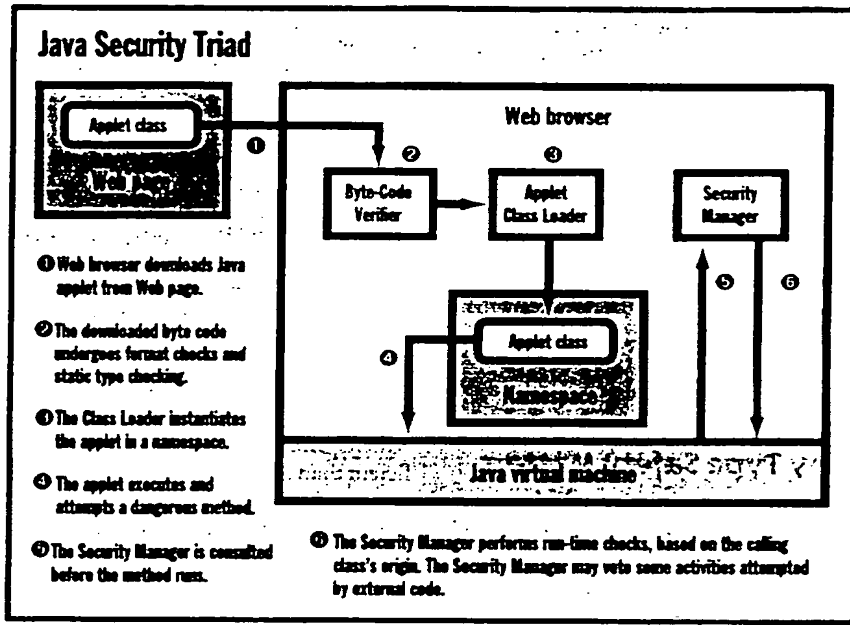
Java security relies on three prongs of defense: the Byte-Code Verifier, the Applet Class Loader, and the Security Manager. Together, these three prongs perform load and run-time checks to restrict file-system and network access, as well as restrict access to browser internals. Each of these prongs depends in some way on the others. Each part must do its job properly for the security model to function correctly.

The Security Triad

The figure "Java Security Triad" above shows how the three prongs of defense fit into the Java framework. The Byte-Code Verifier is the first prong of the Java security model. When a Java source program is compiled, it's converted to platform-independent Java byte code. The Verifier then checks that the untrusted outside code "plays by the rules" before it's allowed to run.

The Verifier checks byte code at a number of different levels. The simplest test ensures that a .class file (i.e., a byte-code file) has the correct format. On a less basic level, the Verifier applies a built-in theorem prover to each method. The theorem prover helps ensure that byte code does not forge pointers, violate access restrictions, or access objects using incorrect type information. The verification process, in concert with the definition of the Java language, helps to establish a base set of security guarantees.

Java's second prong of security defense is the Applet Class Loader. Typically supplied by a browser vendor, it loads all applets and the classes that they reference. When an applet is loaded from the network, the Applet Class Loader receives the binary data and instantiates it as a new class. The Class Loader determines when and how an applet can add classes to a running Java environment. Part of the Class Loader's job is to make sure that the applet doesn't install code that replaces important components of the Java run-time environment.



Java performs several safety checks before a downloaded applet can execute.

In general, a running Java environment can have many active Class Loaders, each defining its own namespace. *Namespaces* allow Java classes to be separated into distinct kinds, according to where they originate. In other words, a namespace is a type-safe portion of memory with classes that are associated with a specific Class Loader.

The third prong of the Java security model is the Security Manager, which restricts the ways in which an applet can use visible interfaces. Thus, the Security

Best Available Copy

BNSDOCID: <XP 679974A I >

Manager implements a good portion of the entire security model. It's a single module that performs run-time checks on dangerous methods, such as those for file or network access or those that define new Class Loaders.

Code in the Java library consults the Security Manager whenever a dangerous operation is about to be attempted. The Security Manager then has a chance to veto the operation by generating a Security Exception. Decisions made by the Security Manager take into account which Class Loader loaded the requesting class. Built-in classes are given more privilege than classes that have been loaded over the network (e.g., applets).

The three parts of the Java security model were created to enforce *type safety*, which means that a program can perform particular operations only on particular kinds of objects. Therefore, Java programs are prevented from accessing memory in inappropriate ways.

More specifically, every piece of memory is part of some Java object, and each object has some class. For example, a calendar management applet might use such classes as Date, Appointment, Alarm, and GroupCalendar. Each class defines a specific set of operations that are allowed to operate on objects of that class. In the calendar management example, the Alarm class might define a `turnOn` operation, but the Date class would not allow `turnOn` to occur.

Why Type Safety Matters

To understand why type safety matters, consider the following, slightly contrived, example. The calendar management applet mentioned above defines a class Alarm, which is represented in memory, as shown in the figure "Type Safety" above. Alarm defines an operation `turnOn`, which sets the first field to true. The Java run-time library defines another class called Applet, whose memory layout is shown in the figure. Note that the first field of Applet is `fileAccessAllowed`, which says whether or not the applet is allowed to access files on the hard disk.

Now suppose that a program tries to apply the `turnOn` operation to an Applet object. If the `turnOn` operation is permitted, the program sets the first field of the object to true. Unfortunately, since the target object is really of type Applet, setting the first field to true allows the applet to access the file system. The applet is then allowed—incorrectly—to modify and even delete files.

How Java Enforces Type Safety

Java labels every object by associating a class tag with it. One simple way to enforce type safety would be to check an object's type tag before every operation on it to make sure that the object's class allows such an operation. This approach is called *dynamic type checking*.

Although this scheme works, it's inefficient. Programs end up running slowly because the system spends a lot of time checking class tags. To improve performance, Java uses static type checking, which is more complicated but more efficient than dynamic type checking. *Static type checking* is where the Java system looks at a program before it runs and carefully deduces the results of the tag-checking operations. If Java can figure out that a particular tag-checking operation will always succeed, then there's no reason to do it. The check can safely be removed, thus speeding up the program.

Java's designers carefully crafted the Java language and byte-code formats to facilitate static type checking. Each piece of byte code is a binary representation of an assembly-like language with op codes and operands.

But Java op codes always take type-specific arguments. There are no "generic" operands that take multiple types in the same operand position, as is the case with processor assembly languages.

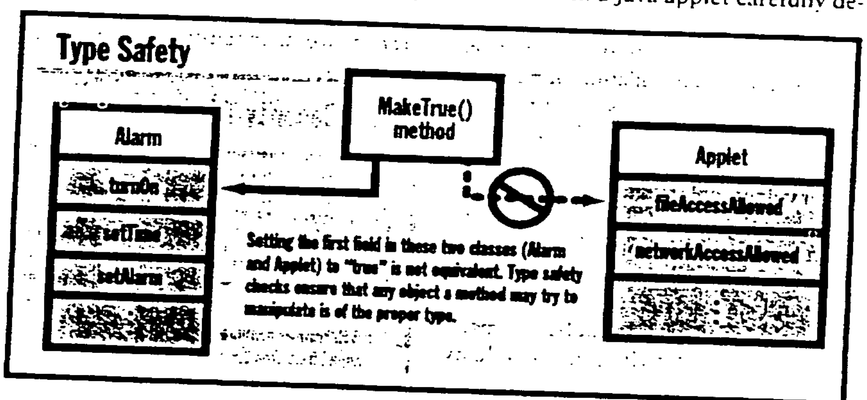
This, and other properties of byte code, make static type checking easier to implement. The Byte-Code Verifier is an effective static type checker that eliminates almost all the tag-checking operations from Java programs. The result is a program that's type-safe but that runs quite efficiently.

Type Confusion

There is only one problem with Java's static type-checking strategy: It's complicated. Although Java's designers obviously got the overall strategy right, a great many details have to be correct for type safety to be enforced. An error in any of these details leaves a tiny, albeit crucial, hole in Java's type-safety armor.

A clever cracker who finds such a hole can launch a *type-confusion attack*. This is done with a Java applet carefully designed to leverage a tiny type-enforcement hole into a complete system penetration. The attacker can set up a situation like the aforementioned Alarm/Applet example, in which the program has one type of object but the Java system thinks the object has some other type.

Because the Verifier normally prohibits such actions, type-confusion errors are usually the result of bugs in the Java implementation. It is hoped that such problems will disappear as the implementation is debugged and refined. ■



Java ensures that malicious programs can't gain access to system resources.

Gary McGraw, Ph.D., is a research scientist at Reliable Software Technologies Corp. (Sterling, VA). He can be reached at **http://www.rstcorp.com/~gem**. *Edward Felten, Ph.D., is an assistant professor of computer science at Princeton University. He can be reached at* **http://www.cs.princeton.edu/~felten**. *Portions of this article are taken from the authors' book* Java Security: Hostile Applets, Holes, and Antidotes *(John Wiley and Sons, 1996).*

Best Available Copy

SDOCID <XP 679974A_ I >

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 718 764 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
26.06.1996 Bulletin 1996/26

(51) Int. Cl.^{6}: G06F 11/00

(21) Application number: 95120052.6

(22) Date of filing: 19.12.1995

(84) Designated Contracting States:
DE FR GB IT NL SE

(30) Priority: 20.12.1994 US 360202

(71) Applicant: SUN MICROSYSTEMS, INC.
Mountain View, CA 94043 (US)

(72) Inventor: Gosling, James A.
Woodside, California 94062 (US)

(74) Representative: Sparing - Röhl - Henseler
Patentanwälte
Rethelstrasse 123
40237 Düsseldorf (DE)

(54) Bytecode program interpreter apparatus and method with pre-verification of data type restrictions

(57) A program interpreter for computer programs written in a bytecode language, which uses a restricted set of data type specific bytecodes. The interpreter, prior to executing any bytecode program, executes a bytecode program verifier procedure that verifies the integrity of a specified program by identifying any bytecode instruction that would process data of the wrong type for such a bytecode and any bytecode instruction sequences in the specified program that would cause underflow or overflow of the operand stack. If the program verifier finds any instructions that violate predefined stack usage and data type usage restrictions, execution of the program by the interpreter is prevented. After pre-processing of the program by the verifier, if no program faults were found, the interpreter executes the program without performing operand stack overflow and underflow checks and without performing data type checks on operands stored in operand stack. As a result, program execution speed is greatly improved.

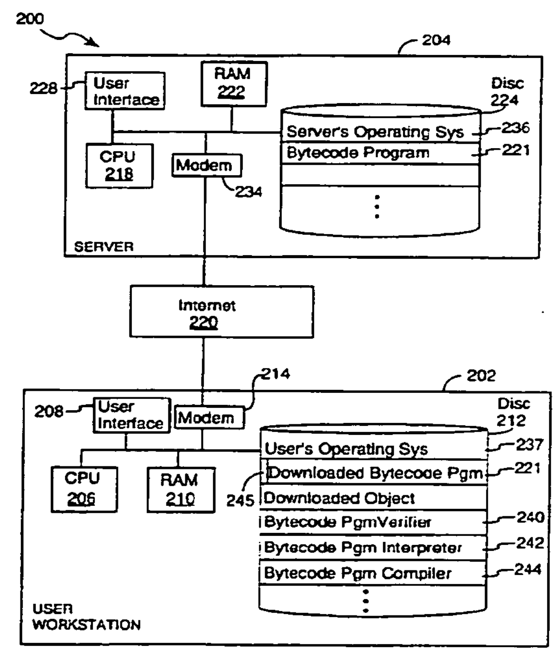


FIGURE 2

EP 0 718 764 A2

Description

BACKGROUND OF THE INVENTION

1. Field of the Invention.

The present invention relates generally to the use of computer software on multiple computer platforms which use distinct underlying machine instruction sets, and more specifically to an efficient program interpreter and method which efficiently handles data type usage checks and operand stack usage checks.

2. Prior Art.

As represented generally in Figure 1, in a typical prior art networked computer system 100, a first computer 102 may download a computer program 103 residing on a second computer 104. In this example, the first user node 102 will typically be a user workstation having a central processing unit 106, a user interface 108, a primary memory 110 (e.g., random access memory) for program execution, a secondary memory 112 (e.g., a hard disc) for storage of an operating system 113, programs, documents and other data, and a modem or other communication interface 114 for connecting to a computer network 120 such as the Internet, a local area network or a wide area network. The computers 102 and 104 are often called "nodes on the network" or "network nodes."

The second computer 104 will often be a network server, but may be a second user workstation, and typically would contain the same basic array of computer components as the first computer.

In the prior art, after the first computer 102 downloads a copy of a computer program 103 from the second computer 104, there are essentially no standardized tools available to help the user of the first computer 102 to verify the integrity of the downloaded program 103. In particular, unless the first computer user studies the source code of the downloaded program, it is virtually impossible using prior art tools to determine whether the downloaded program 103 will underflow or overflow its stack, or whether the downloaded program 103 will violate files and other resources on the user's computer.

A second issue with regard to downloading computer software from one computer to another concerns transferring computer software between computer platforms which use distinct underlying machine instruction sets. There are some prior art examples of platform independent computer programs and platform independent computer programming languages. However, the prior art also lacks tools for efficiently executing such platform independent computer programs while guarding against violation of data type usage restrictions and operand stack usage restrictions.

SUMMARY OF THE INVENTION

The present invention concerns a program interpreter for computer programs written in a bytecode language, to be commercialized as the OAK language, which uses a restricted set of data type specific bytecodes. All the available source code bytecodes in the language either (A) are stack data consuming bytecodes that have associated data type restrictions as to the types of data that can be processed by each such bytecode, (B) do not utilize stack data but affect the stack by either adding data of known data type to the stack or by removing data from the stack without regard to data type, or (C) neither use stack data nor add data to the stack.

The interpreter or the present invention according to a preferred embodiment, prior to executing any bytecode program, executes a bytecode program verifier procedure that verifies the integrity of a specified program by identifying any bytecode instruction that would process data of the wrong type for such a bytecode and any bytecode instruction sequence program that would cause underflow or overflow of the operand stack. If the program verifier finds any instructions that violate pre-defined stack usage and data type usage restrictions, execution of the program by the interpreter is prevented.

The bytecode program verifier aspect of the present invention according to a preferred embodiment includes a virtual operand stack for temporarily storing stack information indicative of data stored in a program operand stack during the execution a specified bytecode program. The verifier processes the specified program by sequentially processing each bytecode instruction of the program, updating the virtual operand stack to indicate the number, sequence and data types of data that would be stored in the operand stack at each point in the program. The verifier also compares the virtual stack information with data type restrictions associated with each bytecode instruction so as to determine if the operand stack during program execution would contain data inconsistent with the data type restrictions of the bytecode instruction, and also determines if any bytecode instructions in the specified program would cause underflow or overflow of the operand stack.

To avoid detailed analysis of the bytecode program's instruction sequence flow, and to avoid verifying bytecode instructions multiple times, all points (called multiple-entry points) in the specified program that can be immediately preceded in execution by two or more distinct bytecodes in the program are identified. Preferably, at least one of the two or more distinct bytecodes in the program will be a jump/branch bytecode. During pre-processing of the specified pro-

NSDOCID <EP 0718764A2 I >

gram, the verifier takes a "snapshot" of the virtual operand stack immediately prior to each multiple-entry point (i.e., subsequent to any one of the preceding bytecode instructions), compares that snapshot with the virtual operand stack state after processing each of the other preceding bytecode instructions for the same multiple-entry point, and generates a program fault if the virtual stack states are not identical.

After pre-processing of the program by the verifier, if no program faults were found, the interpreter executes the program without performing operand stack overflow and underflow checks and without performing data type checks on operands stored in operand stack. As a result, program execution speed is greatly improved.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and form a part of this specification, illustrate embodiments of the invention and, together with the description, serve to explain the principles of the invention, wherein:

Figure 1 depicts two computers interconnected via a network.

Figure 2 depicts two computers interconnected via a network, at least one of which includes a bytecode program verifier in accordance with the present invention.

Figure 3 depicts data structures maintained by a bytecode verifier during verification of a bytecode program in accordance with the present invention.

Figure 4 represents a flow chart of the bytecode program verification process in the preferred embodiment of the present invention.

Figure 5 represents a flow chart of the bytecode program interpreter process in the preferred embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Reference will now be made in detail to the preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. While the invention will be described in conjunction with the preferred embodiments, it will be understood that they are not intended to limit the invention to those embodiments. On the contrary, the invention is intended to cover alternatives, modifications and equivalents, which may be included within the spirit and scope of the invention as defined by the appended claims.

Referring now to a distributed computer system 200 as shown in Figure 2, a first computer node 202 is connected to a second computer node 204 via a computer communications network such as the Internet 220. The first computer node 202 includes a central processing unit 206, a user interface 208, primary memory (RAM) 210, secondary memory (disc storage) 212, and a modem or other communication interface 214 that connects the first computer node 202 to the computer communication network 220. The disc storage 212 stores programs for execution by the processor 206, at least one of which is a bytecode program 221 which is of executable form. For the purposes of this description, it will be assumed that the first computer node 202 receives the bytecode program 221 from the second computer node 204 via the computer communications network 220 using file transfer protocols well known to those skilled in the art.

In the preferred embodiment, the bytecode program is written as an OAK application, which when compiled or interpreted will result in a series of executable instructions. A listing of all the source code bytecode instructions in the OAK instruction set is provided in Table 1. The OAK instruction set is characterized by bytecode instructions that are data type specific. Specifically, the OAK instruction set distinguishes the same basic operation on different primitive data types by designating separate opcodes. Accordingly, a plurality of bytecodes are included within the instruction set to perform the same basic function (for example to add two numbers), with each such bytecode being used to process only data of a corresponding distinct data type. In addition, the OAK instruction set is notable for instructions not included. For instance, there are no "computed goto" instructions in the OAK language instruction set, and there are no instructions for modifying object references or creating new object references (other than copying an existing object reference). These two restrictions on the OAK instruction set, as well as others, help to ensure that any bytecode program which utilizes data in a manner consistent with the data type specific instructions in the OAK instruction set will not violate the integrity of a user's computer system.

In the preferred embodiment, the available data types are integer, long integer, short integer (16 bit signed integer), single precision floating point, double precision floating point, byte, character, and object pointer (sometimes herein called an object reference). The "object reference" data type includes a virtually unlimited number of data subtypes because each "object reference" data type can include an object class specification as part of the data type. In addition, constants used in programs are also data typed, with the available constant data types in the preferred embodiment

BNSDOCID: <EP 0718764A2 I >

comprising the data types mentioned above, plus class, fieldref, methodref, string, and Asciz, all of which represent two or more bytes having a specific purpose.

The few bytecodes that are data type independent perform stack manipulation functions such as (A) duplicating one or more words on the stack and placing them at specific locations within the stack, thereby producing more stack items of known data type, or (B) clearing one or more items from the stack. A few other data type independent bytecode do not utilize any words on the stack and leave the stack unchanged, or add words to the stack without utilizing any of the words previously on the stack. These bytecodes do not have any data type restrictions with regard to the stack contents prior to their execution, and all but a few modify the stack's contents and thus affect the program verification process.

The second computer node 204, assumed here to be configured as a file or other information server, includes a central processing unit 218, a user interface 228, primary memory (RAM) 222, secondary memory (disc storage) 224, and a modem or other communication interface 234 that connects the second computer node to the computer communication network 220. The disc storage 224 stores programs for execution by the processor 218 and/or distribution to other computer nodes.

The first and second computer nodes 202 and 204 may utilize different computer platforms and operating systems 236, 237 such that object code programs executed on either one of the two computer nodes cannot be executed on the other. For instance, the server node 204 might be a Sun Microsystems computer using a Unix operating system while the user workstation node 202 may be an IBM compatible computer using an 80486 microprocessor and a Microsoft DOS operating system. Furthermore, other user workstations coupled to the same network and utilizing the same server 204 might use a variety of different computer platforms and a variety of operating systems.

In the past, a server 204 used for distributing software on a network having computers of many types would store distinct libraries of software for each of the distinct computer platform types (e.g., Unix, Windows, DOS, Macintosh, etc.). Thus, different versions of the same computer program might be stored in each of the libraries. However, using the present invention, many computer programs could be distributed by such a server using just a single, bytecode version of the program.

As shown in Figure 2, the first computer node 202 stores in its secondary memory 212 a bytecode verifier program 240 for verifying the integrity of specified bytecode programs and a bytecode interpreter 242 for executing specified bytecode programs. Alternately, or in addition, the first computer node 202 may store a bytecode compiler 244 for converting a verified bytecode program into an object code program for more efficient execution of the bytecode program 221 than by the interpreter 244.

The bytecode verifier 240 is an executable program which verifies operand data type compatibility and proper stack manipulations in a specified bytecode (source) program 221 prior to the execution of the bytecode program 221 by the processor 206 under the control of the bytecode interpreter 242. Each bytecode program 103 has an associated verification status value 245 that is initially set to False when the program is downloaded from another location. The verification status value 245 for the program is set to True by the bytecode verifier 240 only after the program has been verified not to fail any of the data type and stack usage tests performed by the verifier 240.

During normal execution of a program by an interpreter, the interpreter must continually monitor the operand stack for overflows (i.e., adding more data to the stack than the stack can store) and underflows (i.e., attempting to pop data off the stack when the stack is empty). Such stack monitoring must normally be performed for all instructions that change the stack's status (which includes most all instructions). For many programs, stack monitoring instructions executed by the interpreter account for approximately 80% of the execution time of an interpreted computed program.

In addition, the downloaded bytecode program may contain errors involving the data types of operands not matching the data type restrictions of the instructions using those operands, which may cause the program to be fail during execution. Even worse, a bytecode program might attempt to create object references (e.g., by loading a computed number into the operand stack and then attempting to use the computed number as an object handle) and to thereby breach the security and/or integrity of the user's computer.

Use of the bytecode verifier 240 in accordance with the present invention enables verification of a bytecode program's integrity and allows the use of an interpreter 242 which does not execute the usual stack monitoring instructions during program execution, thereby greatly accelerating the program interpretation process.

The Bytecode Program Verifier

Referring now to Figure 3, the execution of the bytecode program verifier 240 will be explained in conjunction with a particular bytecode program 340. The verifier 240 uses a few temporary data structures to store information it needs during the verification process. In particular, the verifier 240 uses a stack counter 342, a virtual stack 344, a virtual local variable array 345, and a stack snapshot storage structure 346.

The stack counter 342 is updated by the verifier 240 as it keeps track of the virtual stack manipulations so as to reflect the current number of virtual stack 320 entries. The virtual stack 344 stores data type information regarding each datum that will be stored by the bytecode program 340 in the operand stack during actual execution. In the preferred embodiment, the virtual stack 344 is used in the same way as a regular stack, except that instead of storing actual data

BNSDOCID <EP 0718764A2 I >

and constants, the virtual stack 344 stores a data type indicator value for each datum that will be stored in the operand stack during actual execution of the program. Thus, for instance, if during actual execution the stack were to store three values:

HandleToObjectA
5
1

the corresponding virtual stack entries will be

R
I
I

where "R" in the virtual stack indicates an object reference and each "I" in the virtual stack indicates an integer. Furthermore, the stack counter 342 in this example would store a value of 3, corresponding to three values being stored in the virtual stack 344.

Data of each possible data type is assigned a corresponding virtual stack marker value, for instance: integer (I), long integer (L), single precision floating point number (F), double precision floating point number (D), byte (B), short (S), and object reference (R). The marker value for an object reference will often include an object class value (e.g., R:point, where "point" is an object class).

The virtual local variable array 345 serves the same basic function as the virtual stack 344. That is, it is used to store data type information for local variables used by the specified bytecode program. Since data is often transferred by programs between local variables and the operand stack, the bytecode instructions performing such data transfers and otherwise using local variables can be checked to ensure that the local variables accessed by each bytecode instruction are consistent with the data type usage restrictions on those bytecode instructions.

While processing the specified bytecode program, for each datum that would be popped off the stack for processing by a bytecode instruction, the verifier pops off the same number of data type value off the virtual stack 342 and compares the data type values with the data type requirements of the bytecode. For each datum that would be pushed onto the stack by a bytecode instruction, the verifier pushes onto the virtual stack a corresponding data type value.

One aspect of program verification in accordance with present invention is verification that the number and data type of the operands in the operand stack status is identical every time a particular instruction is executed. If a particular bytecode instruction can be immediately preceded in execution by two or more different instructions, then the virtual stack status immediately after processing of each of those different instructions must be compared. Usually, at least one of the different preceding instructions will be a conditional or unconditional jump or branch instruction. A corollary of the above "stack consistency" requirement is that each program loop must not result in a net addition or reduction in the number of operands stored in the operand stack.

The stack snapshot storage structure 346 is used to store "snapshots" of the stack counter 342 and virtual stack 344 to enable efficient comparison of the virtual stack status at various points in the program. Each stored stack snapshot is of the form:

SC, DT1, DT2, DT3, ..., DTn

where SC is the stack counter value, DT1 is the first data type value in the virtual operand stack, DT2 is the second data type value in the virtual operand stack, and so on through DTn which is the data type value for the last possible item in the virtual operand stack.

The stack snapshot storage structure 346 is bifurcated into a directory portion 348 and a snapshot storage portion 350. The directory portion 348 is used to store target instruction identifiers (e.g., the absolute or relative address of each target instruction) while the snapshot portion 350 is used to store virtual stack 344 snapshots associated with the target instruction identifiers.

"Target" instructions are defined to be all bytecode instructions that can be the destination of a jump or branch instruction. For example, a conditional branch instruction includes a condition (which may or may not be satisfied) and a branch indicating to which location (target) in the program the execution is to "jump" in the event the condition is satisfied. In evaluating a conditional jump instruction, the verifier 300 utilizes the stack snapshot storage structure 346 to store both the identity of the target location (in the directory portion 348) and the status of the virtual stack 344 (in the snapshot portion 350) just before the jump. The operation of the stack snapshot storage structure 346 will be explained in greater detail below in conjunction with the description of the execution of the bytecode verifier program.

As was described previously, the bytecode program 350 includes a plurality of data type specific instructions, each of which is evaluated by the verifier 300 of the present invention. The bytecode program 350 includes instructions for stack manipulations 352 and 354 (push integer onto the stack and pop integer from the stack respectively), a forward jump 356 and its associated target 364, a backwards jump 366 and its associated target 362, and a do loop 358 and its associated end 360 (which may be an unconditional or conditional branch instruction, depending on the type of do loop). Since the verifier 240 of the preferred embodiment of the present invention only seeks to verify stack manipulations and data type compatibilities, the operation of the bytecode verifier can be explained using this representative set of instructions.

BNSDOCID <EP 0718764A2 I >

Referring now to Figures 4A-4G, and Appendix 1, the execution of the bytecode verifier program 240 will be described in detail. Appendix 1 lists a pseudocode representation of the verifier program. The pseudocode used in Appendix 1 is, essentially, a computer language using universal computer language conventions. While the pseudocode employed here has been invented solely for the purposes of this description, it is designed to be easily understandable by any computer programmer skilled in the art.

As shown in Figure 4A, the downloaded bytecode program is loaded (400) into the bytecode verifier 300 for processing. The verifier 300 creates (402) the virtual stack 344 and creates the virtual local variable array 345 by designating arrays of locations in memory to store operand and local variable data type information. Similarly, the verifier creates (404) the stack snapshot storage structure by designating an array of locations in memory to store snapshot information. Finally, the verifier designates (406) a register to act as a stack counter 342 for keeping track of the number of virtual stack entries.

A first pass is made through the bytecode program in order to extract target information associated with conditional and un-conditional jumps and loop instructions. In this first pass the verifier 300 sequentially processes all the instructions (steps 408, 410, 412), and for each instruction that is a conditional or unconditional jump (step 414) a representation of the target location for the jump is stored (step 416) in the directory portion 348 of the stack snapshot storage structure 346, unless (step 418) the target location has already been stored in the directory 348. For instance, the absolute or relative address of the target instruction may be stored in the next available slot of the directory 348. All other types of bytecode instructions are ignored on this first pass.

After all the instructions in the program have been processed, the directory 348 is preferably sorted to put the target locations noted in the directory in address sequential order.

Referring again to Figure 3, for the purposes illustration the stack snapshot storage structure 346 has been loaded with the information which would have been stored in the directory portion 348 as if the first pass of the verification had been completed based on the bytecode instructions shown in bytecode program 350. Specifically, the directory portion has been loaded with the addresses associated with all of the targets of the conditional and unconditional jumps resident in the bytecode program.

Referring now to Figure 4B, a second pass through the bytecode program is initiated in order to verify proper use of the operand stack and of data types by the bytecode program. The first instruction of the bytecode program is selected (430) and the verifier first checks (432) to see if the address for the selected instruction has been stored in the directory portion 348 of the stack snapshot storage structure 346 in the first pass described above.

If the address of the selected instruction is in the directory 348, indicating that the selected instruction is the target of a conditional or un-conditional jump, the verifier checks (434) to see if an associated stack snapshot has been stored in the snapshot portion 350 of the stack snapshot storage structure 346. If a stack snapshot has not been stored (indicating that the instruction is a target of a backward jump), then the contents of the virtual stack and the stack counter are stored (436) in the stack snapshot storage structure 346. The snapshot contains information on the status of the virtual stack just before the execution of the instruction being processed, including a data type value for each datum that has been pushed onto the stack.

If a stack snapshot has been stored for the currently selected instruction (indicating that a jump instruction associated with this target instruction has already been processed), then the verifier compares (438) the virtual stack snapshot information stored in the snapshot portion 350 of the stack snapshot storage structure 346 for the currently selected instruction with the current state of the virtual stack. If the comparison shows that the current state and the snapshot do not match, then an error message or signal is generated (440) identifying the place in the bytecode program where the stack status mismatch occurred. In the preferred embodiment, a mismatch will arise if the current virtual stack and snapshot do not contain the same number or types of entries. The verifier will then set a verification status value 245 for the program to false, and abort (442) the verification process. Setting the verification status value 245 for the program to false prevents execution of the program by the bytecode interpreter 242 (Figure 2).

If the current virtual stack and the stored stack snapshot for the current instruction match (438), then the verifier will continue the verification process and analyze the individual instruction, starting at step 450, as described below.

If the address of the currently selected instruction is not found within the directory portion 348 of the stack snapshot storage structure 346 or if a stack status mismatch is not detected, then the verifier performs selected ones of a series of checks on the instruction depending on the particular instructions stack usage and function.

Referring to Figure 4C, the first check to be performed concerns instructions that pop data from the operand stack. If the currently selected instruction pops data from the stack (450), the stack counter is inspected (452) to determine whether there is sufficient data in the stack to satisfy the data pop requirements of the instruction.

If the operand stack has insufficient data (452) for the current instruction, that is called a stack underflow, in which case an error signal or message is generated (454) identifying the place in the program that the stack underflow was detected. In addition, the verifier will then set a verification status value 245 for the program to false, and abort (456) the verification process.

If no stack underflow condition is detected, the verifier will compare (458) the data type code information previously stored in the virtual stack with the data type requirements (if any) of the currently selected instruction. For example, if

BNSDOCID <EP 0718764A2 I >

the opcode of the instruction being analyzed calls for an integer add of a value popped from the stack, the verifier will compare the operand information of the item in the virtual stack which is being popped to make sure that is of the proper data type, namely integer. If the comparison results in a match, then the verifier deletes (460) the information from the virtual stack associated with the entry being popped and updates the stack counter 342 to reflect the number of entries popped from the virtual stack 344.

If a mismatch is detected (458) between the stored operand information in the popped entry of the virtual stack 344 and the data type requirements of the currently selected instruction, then a message is generated (462) identifying the place in the bytecode program where the mismatch occurred. The verifier will then set a verification status value 245 for the program to false and abort (456) the verification process. This completes the pop verification process.

Referring to Figure 4D, if the currently selected instruction pushes data onto the stack (470), the stack counter is inspected (472) to determine whether there is sufficient room in the stack to store the data the selected instruction will push onto the stack. If the operand stack has insufficient room to store the data to be pushed onto the stack by the current instruction (472), that is called a stack overflow, in which case an error signal or message is generated (474) identifying the place in the program that the stack underflow was detected. In addition, the verifier will then set a verification status value 245 for the program to false, and abort (476) the verification process.

If no stack overflow condition is detected, the verifier will add (478) an entry to the virtual stack indicating the type of data (operand) which is to be pushed onto the operand stack (during the actual execution of the program) for each datum to be pushed onto the stack by the currently selected instruction. This information is derived from the data type specific opcodes utilized in the bytecode program of the preferred embodiment of the present invention. The verifier also updates the stack counter 342 to reflect the added entry or entries in the virtual stack. This completes the stack push verification process.

Referring to Figure 4E, if the currently selected instruction causes a conditional or unconditional jump or branch forward in the program beyond the ordinary sequential step operation (step 480) the verifier will first check (482) to see if a snapshot for the target location of the jump instruction is stored in the stack snapshot storage structure 346. If a stack snapshot has not been stored, then the virtual stack configuration (subsequent to any virtual stack updates associated with the jump) is stored (484) in the stack snapshot storage structure 346 at a location associated with the target program location. Note that any stack pop operations associated with the jump will have already been reflected in the virtual stack by the previously executed step 460 (see Figure 4C).

If a stack snapshot has been stored (indicating that another entry point associated with this target instruction has already been processed), then the verifier compares (486) the virtual stack snapshot information stored in the snapshot portion 340 of the stack snapshot storage structure 346 with the current state of the virtual stack. If the comparison shows that the current state and the snapshot do not match, then an error message is generated (488) identifying the place in the bytecode program where the stack status mismatch occurred. In the preferred embodiment, a mismatch will arise if the current virtual stack and snapshot do not contain the same number or types of entries. Furthermore, a mismatch will arise if one or more data type values in the current virtual stack do not match corresponding data type values in the snapshot. The verifier will then set a verification status value 245 for the program to false and abort (490) the verification process. If a stack status match is detected at step 486, then the verifier continues processing at step 500 (Figure 4F).

Referring to Figure 4F, if the currently selected instruction causes a conditional or unconditional jump or branch backward in the program (step 500) then the verifier compares (502) the virtual stack snapshot information stored in the snapshot portion 340 of the stack snapshot storage structure 346 associated with the target of the backward jump (which has already been stored in step 436) with the current state of the virtual stack. If the comparison shows that the current state and the snapshot do not match, then an error message is generated (504) identifying the place in the bytecode program where the stack status mismatch occurred. In the preferred embodiment, a mismatch will arise if the current virtual stack and snapshot do not contain the same number or types of entries or if any data type entry in the current virtual stack does not match the corresponding data type entry in the snapshot. The verifier will then set a verification status value 245 for the program to false and abort (506) the verification process.

If a stack status match is detected (at step 502) or if the instruction is not a backward jump (at step 500), then the verifier continues processing at step 510.

If the currently selected instruction reads data from a local variable (510), the verifier will compare (512) the data type code information previously stored in the corresponding virtual local variable with the data type requirements (if any) of the currently selected instruction. If a mismatch is detected (512) between the data type information stored in the virtual local variable and the data type requirements of the currently selected instruction, then a message is generated (514) identifying the place in the bytecode program where the mismatch occurred. The verifier will then set a verification status value 245 for the program to false and abort (516) the verification process.

If the currently selected instruction does not read data from a local variable (510) or the data type comparison at step 512 results in a match, then the verifier continues processing the currently selected instruction at step 520.

Referring to Figure 4G, if the currently selected instruction stores data into a local variable (520), the corresponding virtual local variable is inspected (522) to determine whether it stores a data type value. If the virtual local variable does

BNSDOCID <EP 0718764A2 I >

store a data type value (indicating that data has been previously stored in the local variable), the verifier compares the data type information in the virtual local variable with the data type associated with the currently selected bytecode instruction (524). If a mismatch is detected (524) between the data type information stored in the virtual local variable and the data type requirements of the currently selected instruction, then a message is generated (526) identifying the place in the bytecode program where the mismatch occurred. The verifier will then set a verification status value 245 for the program to false and abort (528) the verification process.

If the currently selected instruction does not store data into a local variable (520) processing for the currently selected instruction is completed. If the currently selected instruction stores data into a local variable, but the virtual local variable does not store a data type value (indicating that no instruction which would store data in the local variable has yet been processed by the verifier), then the data type associated with the selected bytecode instruction is stored in the virtual local variable (step 530).

Next, the verifier checks (540) to see if this is the last instruction in the bytecode program 340 to be processed. If more instructions remain to be processed, then the verifier loads (542) the next instruction, and repeats the verification process starting at step 432. If no more instructions are to be processed, then the verifier will then set a verification status value 245 for the program to True (544), signaling the completion of the verification process.

Bytecode Interpreter

Referring to flow chart in Figure 5 and Appendix 2, the execution of the bytecode interpreter 242 will be described. Appendix 2 lists a pseudocode representation of the bytecode interpreter.

After a specified bytecode program has been received or otherwise selected (560) as a program to be executed, the bytecode program interpreter 242 calls (562) the bytecode verifier 240 to verify the integrity of the specified bytecode program. The bytecode verifier is described above.

If the verifier returns a "verification failure" value (564), the attempt to execute the specified bytecode program is aborted by the interpreter (566).

If the verifier 242 returns a "Verification Success" value (564), the specified bytecode program is linked (568) to resource utility programs and any other programs, functions and objects that may be referenced by the program. Such a linking step is a conventional pre-execution step in many program interpreters. Then the linked bytecode program is interpreted and executed (570) by the interpreter. The bytecode interpreter of the present invention does not perform any operand stack overflow and underflow checking during program execution and also does not perform any data type checking for data stored in the operand stack during program execution. These conventional stack overflow, underflow and data type checking operations can be skipped by the present invention because the interpret has already verified that errors of these types will not be encountered during program execution.

The program interpreter of the present invention is especially efficient for execution of bytecode programs having instruction loops that are executed many times, because the operand stack checking instructions are executed only once for each bytecode in each such instruction loop in the present invention. In contrast, during execution of a program by a convention interpreter, the interpreter must continually monitor the operand stack for overflows (i.e., adding more data to the stack than the stack can store) and underflows (i.e., attempting to pop data off the stack when the stack is empty). Such stack monitoring must normally be performed for all instructions that change the stack's status (which includes most all instructions). For many programs, stack monitoring instructions executed by the interpreter account for approximately 80% of the execution time of an interpreted computed program. As a result, the interpreter of the present invention will often execute programs at two to five times the speed of a conventional program interpreter running on the same computer.

The foregoing descriptions of specific embodiments of the present invention have been presented for purposes of illustration and description. They are not intended to be exhaustive or to limit the invention to the precise forms disclosed, and obviously many modifications and variations are possible in light of the above teaching. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, to thereby enable others skilled in the art to best utilize the invention and various embodiments with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the Claims appended hereto and their equivalents.

NSDOCID: <EP 0718764A2 I >

TABLE 1
BYTECODES IN OAK LANGUAGE

INSTRUCTION NAME	SHORT DESCRIPTION
aaload	load object reference from array
aastore	store object reference into object reference array
aconst_null	push null object
aload	load local object variable
areturn	return object reference from function
arraylength	get lenth of array
astore	store object reference into local variable
astore_<n>	store object reference into local variable
athrow	throw exception
bipush	push one-byte signed integer
breakpoint	call breakpoint handler
catchsetup	set up exception handler
catchteardown	reset exception handler
checkcast	make sure object is of a given type
df2	convert double floating point number to single precision floating point number
d2i	convert double floating point number to integer
d2l	convert double floating point number to long integer
dadd	add double floating point numbers
daload	load double floating point number from array
dastore	store double floating point number into array
dcmpg	compare two double floating point numbers (return 1 on incomparable)
dcmpl	compare two double floating point numbers (return -1 on incomparable)
dconst_<d>	push double floating point number
ddiv	divide double floating point numbers
dload	load double floating point number from local variable
dload_<n>	load double floating point number from local variable
dmod	perform modulo function on double floating point numbers

BNSDOCID <EP _ 0718764A2 I >

dmul	miltiply double floating point numbers
dneg	negate double floating point number
dreturn	return double floating point number from function
dstore	store double floating point number into local variable
dstore_<n>	store double floating point number into local variable
dsub	subtract double floating point numbers
dup	duplicate top stack word
dup2	duplicate top two stack words
dup2_x1	duplicate top two stack words and put two down
dup2_x2	duplicate top two stack words and put three down
dup_x1	dulicate top stack word and put two down
dup_x2	duplicate top stack word and put three down
f2d	convert single precision floating point number to double floating point number
f2i	convert single precision floating point number to integer
f2l	convert Single precision floating point number to long integer
fadd	add single precision floating point numbers
faload	load single precision floating point number from array
fastore	store into single precision floating point number array
fempg	compare single precision floating point numbers (return 1 on incomparable)
fempl	compare Single precision floating point number (return -1 on incomparable)
fconst_<f>	push single precision floating point number
fdiv	divide single precision floating point numbers
fload	load single precision floating point number from local variable
fload_<n>	load single precision floating point number from local variable
fmod	perform modulo function on single precision floating point numbers
fmul	multiply single precision floating point numbers

NSDOCID: <EP 0718764A2 I >

fneg	negate single precision floating point number
freturn	return single precision floating point number from function
fstore	store single precision floating point number into local variable
fstore_<n>	store single precision floating point number into local variable
fsub	subtract single precision floating point numbers
getfield	fetch field from object
getstatic	set static field from class
goto	branch always
i2d	convert integer to double floating point number
i2f	convert integer to single precision floating point number
i2l	convert integer to long integer
iadd	add integers
iaload	load integer from array
iand	boolean AND two integers
iastore	store into integer array
iconst_<n>	push integer
iconst_m1	push integer constant minus 1
idiv	integer divide
if_acmpeq	branch if objects same
if_acmpne	branch if objects not same
if_icmpeq	branch if integers equal
if_icmpge	branch if integer greater than or equal to
if_icmpgt	branch if integer greater than
if_icmple	branch if integer less than or equal to
if_icmplt	branch if integer less than
if_icmpne	branch if integers not equal
ifeq	branch if equal to 0
ifge	branch if greater than or equal to 0
ifgt	branch if greater than 0
ifle	branch if less than or equal to 0
iflt	branch if less than 0
ifne	branch if not equal to 0
iinc	increment local variable by constant
iload	load integer from local variable

BNSDOCID <EP 0718764A2 I >

iload_<n>	load integer from local variable
imod	peform modulo function on integers
imul	multiply integers
ineg	negate integer
instanceof	determine if object is of given type
int2byte	convert integer to signed byte
int2char	convert integer to char
invokeinterface	invoke interface method
invokemethod	invoke class method
invokesuper	invoke superclass method
ior	boolean OR two integers
iretum	retum integer from function
ishl	integer shift left
Ishr	integer arithmetic shift right
istore	store integer into local variable *vindex*
istore_<n>	store integer into local variable n
isub	subtract integers
iushr	integer logical shift right
ixor	boolean XOR two integers
jsr	jump to subroutine
12d	convert long integer into double floating point number
12f	convert long integer into single precision floating point number
12i	convert long integer into integer
ladd	add long integers
laload	load long integer from array
land	boolean AND two long integers
lastore	store into long integer array
lcmp	compare long integers
lconst_<l>	push long integer constant
ldc1	push item from constant pool
ldc2	push item from constant pool
ldc2w	push long or double from constant pool
ldiv	divide long integers
lload	load long integer from local variable
lload_<n>	load long integer from local variable
lmod	perform modulo function on long integers

BDOCID: <EP__ 0718764A2_I_>

lmul	multiply long integers
lneg	Negate long integer
lookupswitch	Access jump table by key match and jump
lor	boolean OR two long integers
lreturn	return long integer from function
lshl	long integer shift left
lshr	long integer arithmetic shift right
lstore	store long integer into local variable
lstore_<n>	store long integer into local variable
lsub	subract long integers
lushr	long integer logical shift right
lxor	boolean XOR long integers
monitorenter	enter monitored region of code
monitorexit	exit monitored region of code
new	create new object
newarray	allocate new array
newfromname	create new object from name
nop	do nothing
pop	pop top stack word
pop2	pop top two stack words
putfield	set field in object
putstatic	set static field in class
ret	return from subroutine
return	return (void) from procedure
saload	load signed byte from array
sastore	store into signed byte array
siaload	load unsigned short from array
siastore	store into unsigned short array
sipush	push two-byte signed integer
tableswitch	access jump table by index and jump
verifystack	verify stack empty

BNSDOCID <EP 0718764A2 I >

APPENDIX 1

Pseudocode for OAK Bytecode Verifier

```
Receive Bytecode Program to be verified.
Create Virtual Operand Stack Data Structure for storing stack status
information and Virtual Local Variable Array for storing local variable data
type information.
Create data structure for storing Virtual Stack Snapshots.

First Pass through Bytecode Program:
    Locate all instructions that are the targets of conditional and
    unconditional jumps or branches (i.e., can be entered from more than one
    prior instruction).

    Store list of such target instructions in Virtual Stack Snapshot data
    structure.

Second Pass through Bytecode Program:
    Set VerificationSuccess to True
    Do Until Last Bytecode Instruction has been processed:
        {
        Select next bytecode instruction (in sequential order in program)
        If instruction is in list of target instructions
            {
            If snapshot of virtual stack for this instruction already exists
                {
                Compare current state of virtual stack with stored snapshot
                If snapshot does not match current virtual stack state
                    {
                    Print message identifying place in program that stack
                    mismatch occurred
                    Abort Verification
                    Set VerificationSuccess to False
                    Return
                    }
                }
            Else
                Store snapshot of current virtual stack status
```


NSDOCID <EP 0718764A2 I >

```
        }

    Case(Instruction Type):
        {
        Case=Instruction pops data from Operand Stack
            {
            Check for Stack Underflow
            If Stack has Underflowed
                {
                Print message identifying place in program that
                underflow occurred
                Abort Verification
                Return
                }
            Compare data type of each operand popped from stack with
            data type required (if any) by the bytecode instruction
            If type mismatch
                {
                Print message identifying place in program that data type
                mismatch occurred
                Set VerificationSuccess to False
                }
            Delete information from Virtual Stack for popped operands
            Update Stack Counter
            }

        Case=Instruction pushes data onto Operand Stack
            {
            Check for Stack Overflow
            If Stack has Overflowed
                {
                Print message identifying place in program that overflow
                    occurred
                Abort Verification
                Set VerificationSuccess to False
                Return
                }
```


BNSDOCID <EP 0718764A2 I >

```
        Add information to Virtual Stack indicating data type of data
        pushed onto operand stack
        Update Stack Counter
        }

    Case=Instruction is a forward jump or branch instruction
      {
      If snapshot of virtual stack for the target instruction already
      exists
          {
          Compare current state of virtual stack with stored
              snapshot
          If snapshot does not match current virtual stack state
              {
              Print message identifying place in program that stack
              mismatch occurred
              Abort Verification
              Set VerificationSuccess to False
              Return
              }
          }
      Else
          Store snapshot of current virtual stack state as snapshot
          for the target instruction;
      }

  Case=Instruction is an end of loop backward jump or other
  backward jump or branch instruction:
      {
      Compare current virtual stack state with stored snapshot for
      target instruction
      If current virtual stack state does not match stored snapshot
          {
          Print message identifying place in program that stack
          mismatch occurred
          Abort Verification
          Set VerificationSuccess to False
          Return
```


NSDOCID: <EP _ 0718764A2 I >

```
            }
        }

    Case=Instruction reads data from local variable
        {

        Compare data type of each datum read from local variable
        with data type required (if any) by the bytecode instruction
        If type mismatch
            {
            Print message identifying place in program that data type
            mismatch occurred
            Set VerificationSuccess to False
            }
        }

    Case=Instruction stores data into a local variable
        {
        If corresponding virtual local variable already stores a data
        type value
            {
            Compare data type value stored in virtual local variable
            with data type of datum that would be stored in the
            corresponding local variable (as determined by the data
            type handled by the current bytecode instruction)
            If type mismatch
                {
                Print message identifying place in program that data
                type mismatch occurred
                Set VerificationSuccess to False
                }
            }
        Else
            Add information to Virtual Local Variable indicating data
            type of data that would be stored in corresponding local
            variable
        }
```


BNSDOCID <EP 0718764A2 I >

```
} /* EndCase */
} /* End of Do Loop */
Return (VerificationSuccess)
```

APPENDIX 2

Pseudocode for Bytecode Interpreter

```
Receive Specified Bytecode Program to be executed
Call Bytecode Verifier to verify Specified Bytecode Program
If Verification Success
    {
    Link Specified Bytecode Program to resource utility programs.

    Interpret and execute Specified Bytecode Program instructions without
    performing operand stack overflow and underflow checks and without
    performing data type checks on operands stored in operand stack.
    }
```

Claims

1. A method of operating a computer system, the steps of the method comprising:

   (A) storing a program in a memory, the program including a sequence of bytecodes, where each of a multiplicity of said bytecodes each represents an operation on data of a specific data type; said each bytecode having associated data type restrictions on the data type of data to be manipulated by said each bytecode;
   (B) prior to execution of said program, preprocessing said program by determining whether execution of any bytecode in said program would violate said data type restrictions for that bytecode and generating a program fault signal when execution of any bytecode in said program would violate the data type restrictions for that bytecode;
   (C) when said preprocessing of said program results in the generation of no program fault signals, enabling execution of said program; and
   (D) when said preprocessing of said program results in the generation of a program fault, preventing execution of said program.

2. The method of claim 1, said preprocessing step including:

   (B1) determining the state of a virtual stack associated with said program before and after execution of each said bytecode in the program, said virtual stack state storing data type values for operands that would be stored in an operand stack during execution of said program; and

NSDOCID: <EP 0718764A2 I >

(B2) determining whether execution of any bytecode in said program would violate said data type restrictions for that bytecode and generating a program fault signal when execution of any bytecode in said program would violate the data type restrictions for that bytecode.

3. The method of claim 2,
said bytecode program including at least one execution loop;
said preprocessing step including

(B3) determining whether execution of any loop in said program would result in a net addition or deletion of operands to said operand stack, and for generating a program fault signal when execution of any loop in said program would produce a net addition or deletion of operands to said operand stack.

4. The method of claim 3, including
when execution of said bytecode program has been enabled, executing said bytecode program without performing operand stack underflow and overflow checks during execution of said bytecode program.

5. The method of claim 1,2, 3 or 4, including:
when execution of said bytecode program has been enabled, executing said bytecode program without performing data type checks on operands stored in said operand stack during execution of said bytecode program.

6. The method of claim 1,
said bytecode program including at least one execution loop;
said step (B) including
determining the state of a virtual stack associated with said program before and after execution of each said bytecode in the program, said virtual stack state storing data type values for operands that would be stored in an operand stack during execution of said program; and
determining whether execution of any loop in said program would result in a net addition or deletion of operands to said operand stack, and for generating a program fault signal when execution of any loop in said program would produce a net addition or deletion of operands to said operand stack.

7. The method of claim 6, including
when execution of said bytecode program has been enabled, executing said bytecode program without performing operand stack underflow and overflow checks during execution of said bytecode program.

8. The method of claim 1,
said step (B) including determining, whenever two or more bytecodes in said program comprise jumps/branches to an identical location in said program, whether the states of the virtual stack prior to execution of each of said jump/branches are inconsistent, and for generating a program fault signal if said virtual stack states are inconsistent.

9. The method of claim 8,
when execution of said bytecode program has been enabled, executing said bytecode program without performing operand stack status checks during execution of said bytecode program.

10. A computer system, comprising:
memory for storing a bytecode program, the bytecode program including a sequence of bytecodes, where each of a multiplicity of said bytecodes each represents an operation on data of a specific data type; said each bytecode having associated data type restrictions on the data type of data to be manipulated by said each bytecode;
a data processing unit for executing programs stored in said memory;
a bytecode program verifier, stored in said memory, said bytecode program verifier including data type testing instructions for determining whether execution of any bytecode in said program would violate said data type restrictions for that bytecode and generating a program fault signal when execution of any bytecode in said program would violate the data type restrictions for that bytecode; and
a bytecode program interpreter, coupled to said bytecode program verifier, that executes said bytecode program after processing of said bytecode program by said bytecode program verifier only when said bytecode program verifier generates no program fault signals.

11. The computer system of claim 10,
said bytecode program verifier including

BNSDOCID: <EP 0718764A2 I >

stack status tracking instructions for determining the state of a virtual stack associated with said program before and after execution of each said bytecode in the program, said virtual stack state storing data type values for operands that would be stored in an operand stack during execution of said program; and

data type checking instructions for determining whether execution of any bytecode in said program would violate said data type restrictions for that bytecode and generating a program fault signal when execution of any bytecode in said program would violate the data type restrictions for that bytecode.

12. The computer system of claim 10, said bytecode program verifier further including:

stack overflow/underflow testing instructions for determining whether execution of said program would result in an operand stack underflow or overflow and generating a program fault signal when execution of said program would result in an operand stack underflow or overflow.

13. The computer system of claim 12, said bytecode program interpreter including instructions for executing said bytecode program without performing operand stack underflow and overflow checks during execution of said bytecode program.

14. The computer system of claim 10, 11, 12, or 13 said bytecode program interpreter including instructions for executing said bytecode program without performing data type checks on operands used by said bytecode program.

15. The computer system of claim 10,

said bytecode program verifier including

stack status tracking instructions for determining the state of a virtual stack associated with said program before and after execution of each said bytecode in the program, said virtual stack state storing data type values for operands that would be stored in an operand stack during execution of said program; and

stack overflow/underflow testing instructions for determining whether execution of said program would result in an operand stack underflow or overflow and for generating a program fault signal when execution of said program would result in an operand stack underflow or overflow.

16. The computer system of claim 10,

said bytecode program interpreter including means for executing said bytecode program without performing operand stack underflow and overflow checks during execution of said bytecode program.

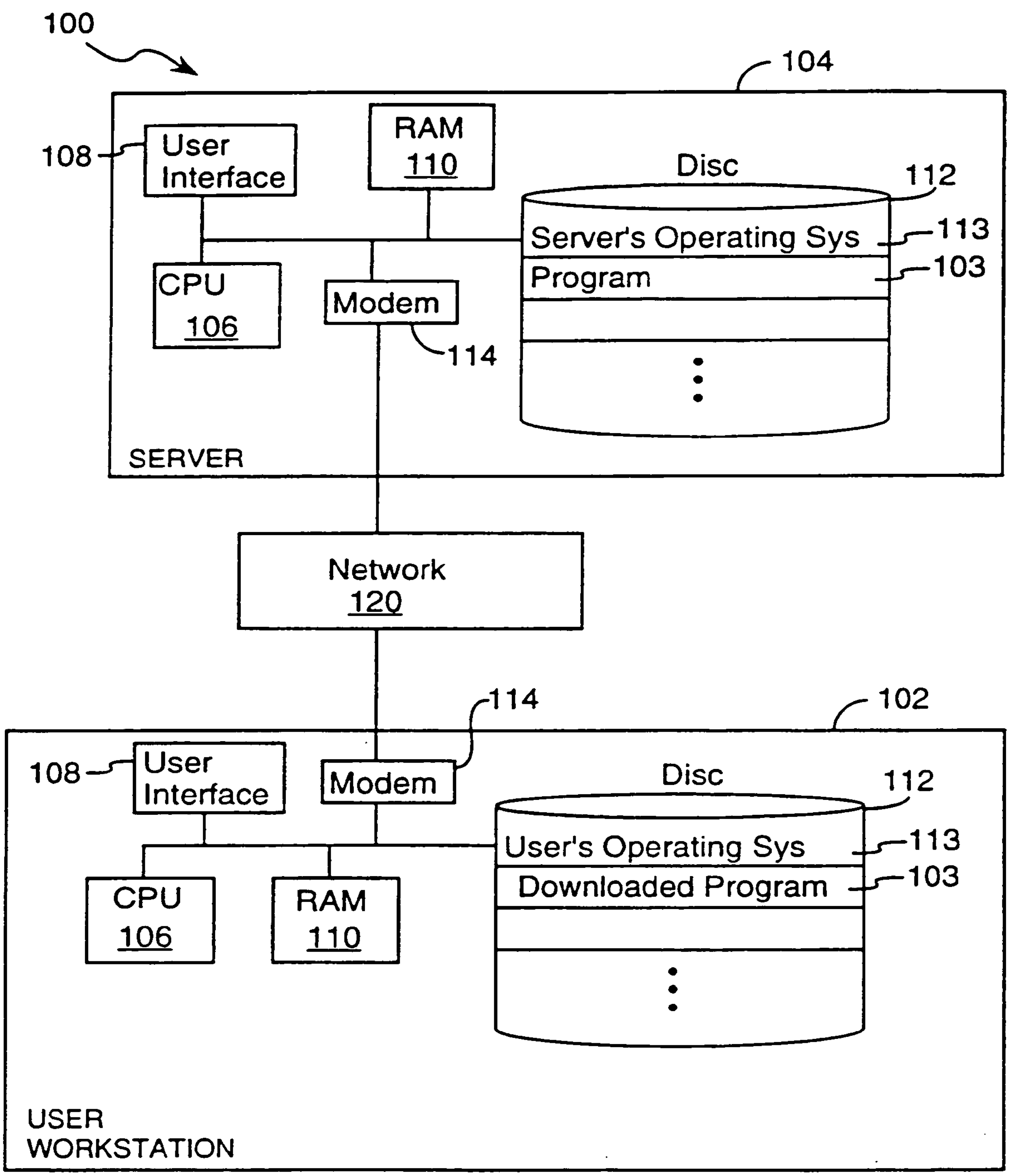
17. The computer system of claim 10,

said bytecode program verifier including jump/branch inspection instructions for determining, whenever two or more bytecodes in said program comprise jumps/branches to an identical location in said program, whether the states of the virtual stack prior to execution of each of said jump/branches are inconsistent, and for generating a program fault signal if said virtual stack states are inconsistent.

18. The computer system of claim 17,

said bytecode program interpreter including means for executing said bytecode program without performing operand stack status checks during execution of said bytecode program.

NSDOCID: <EP 0718764A2 I >



Prior Art

FIGURE 1

BNSDOCID: <EP 0718764A2 I >

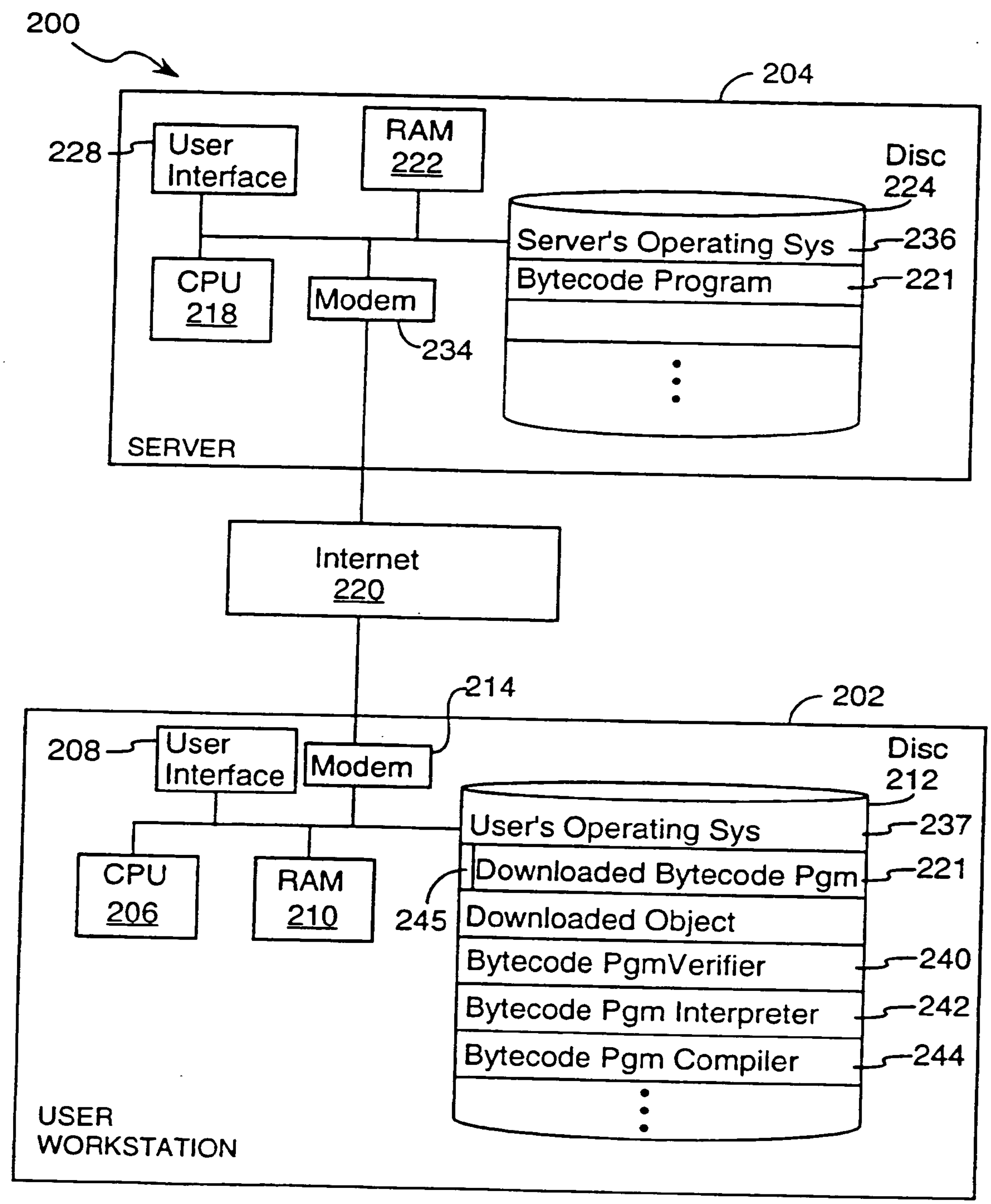


FIGURE 2

BNSDOCID: <EP 0718764A2 I >

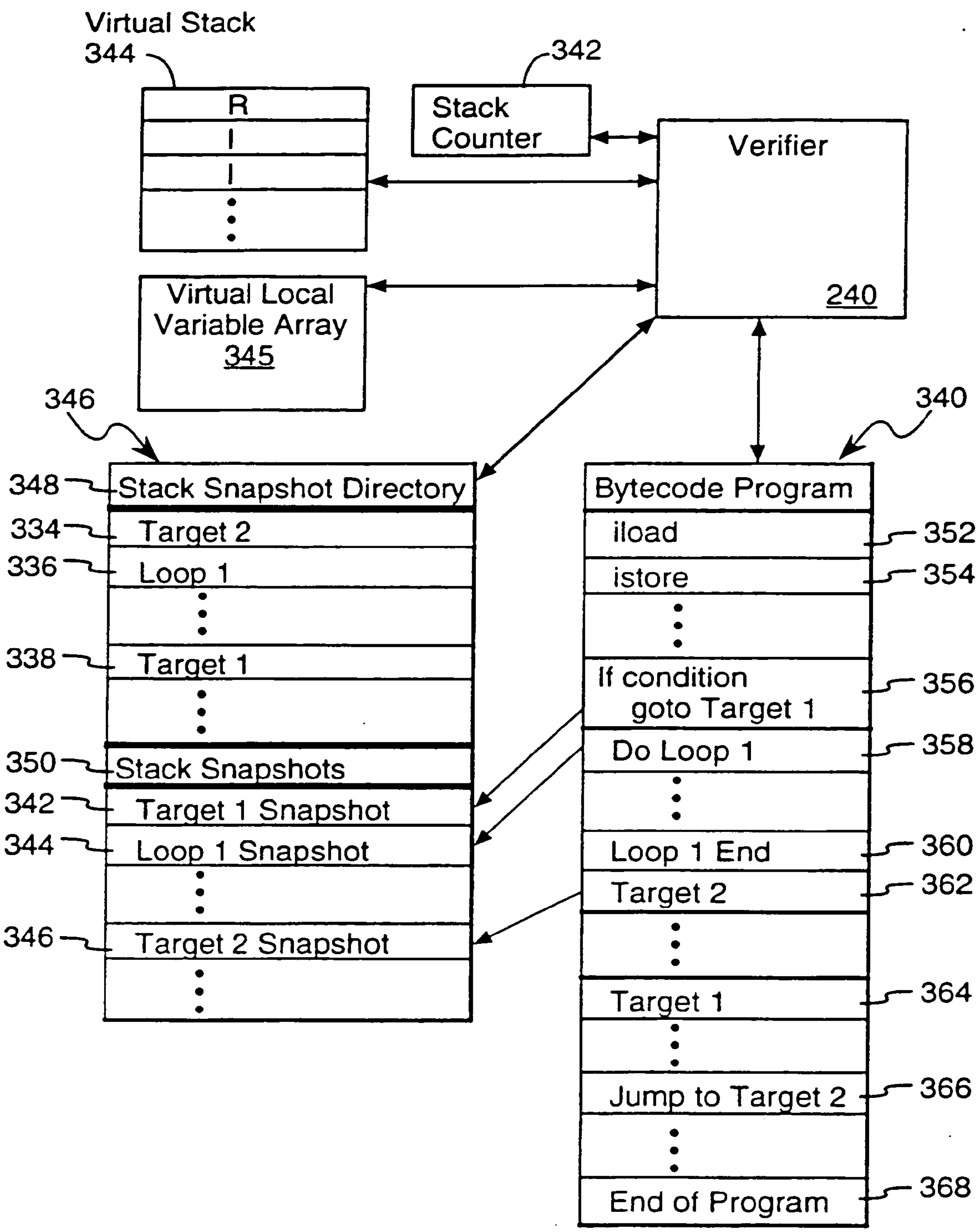


FIGURE 3

BNSDOCID: <EP 0718764A2 I >

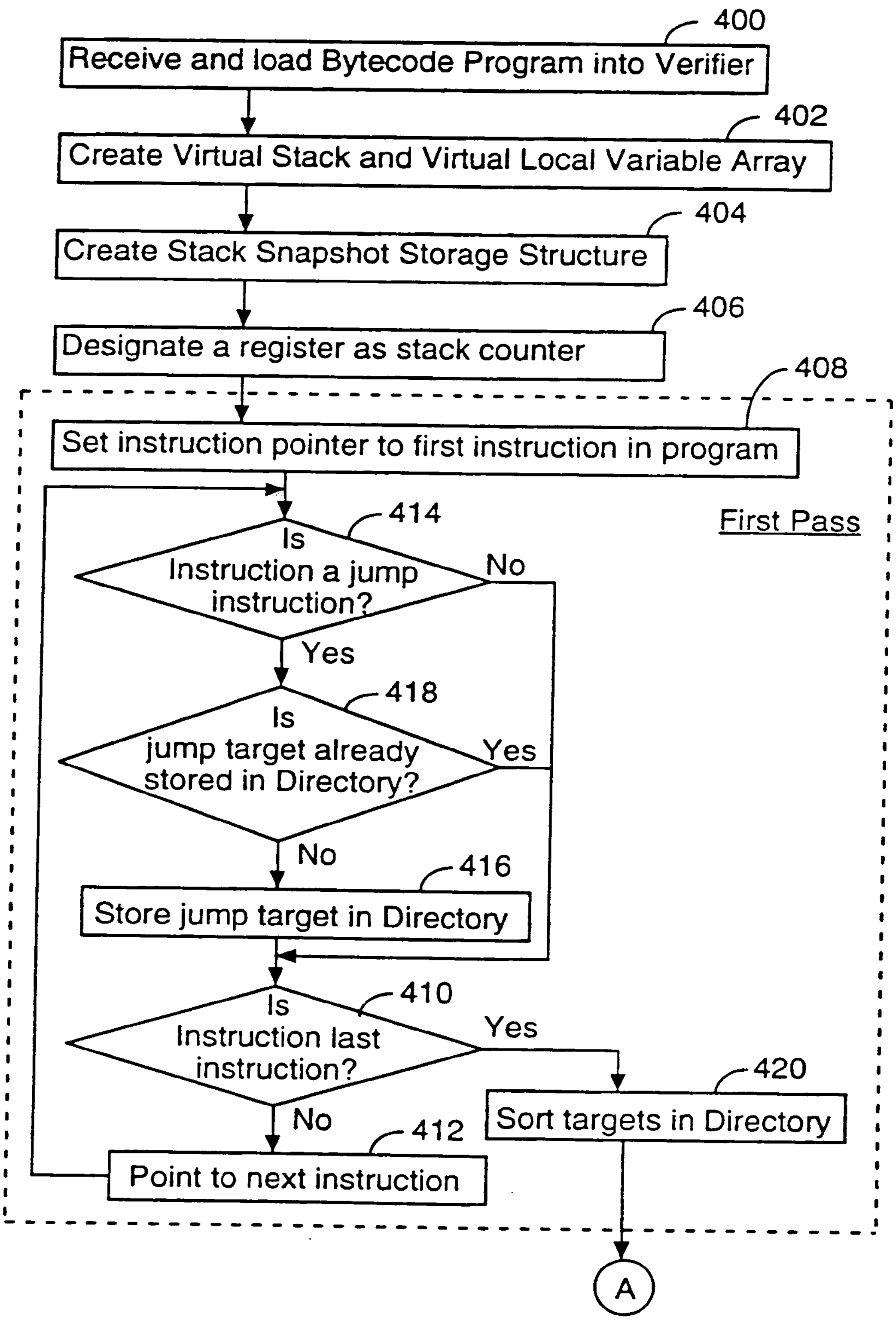


FIGURE 4A

NSDOCID: <EP 0718764A2 I >

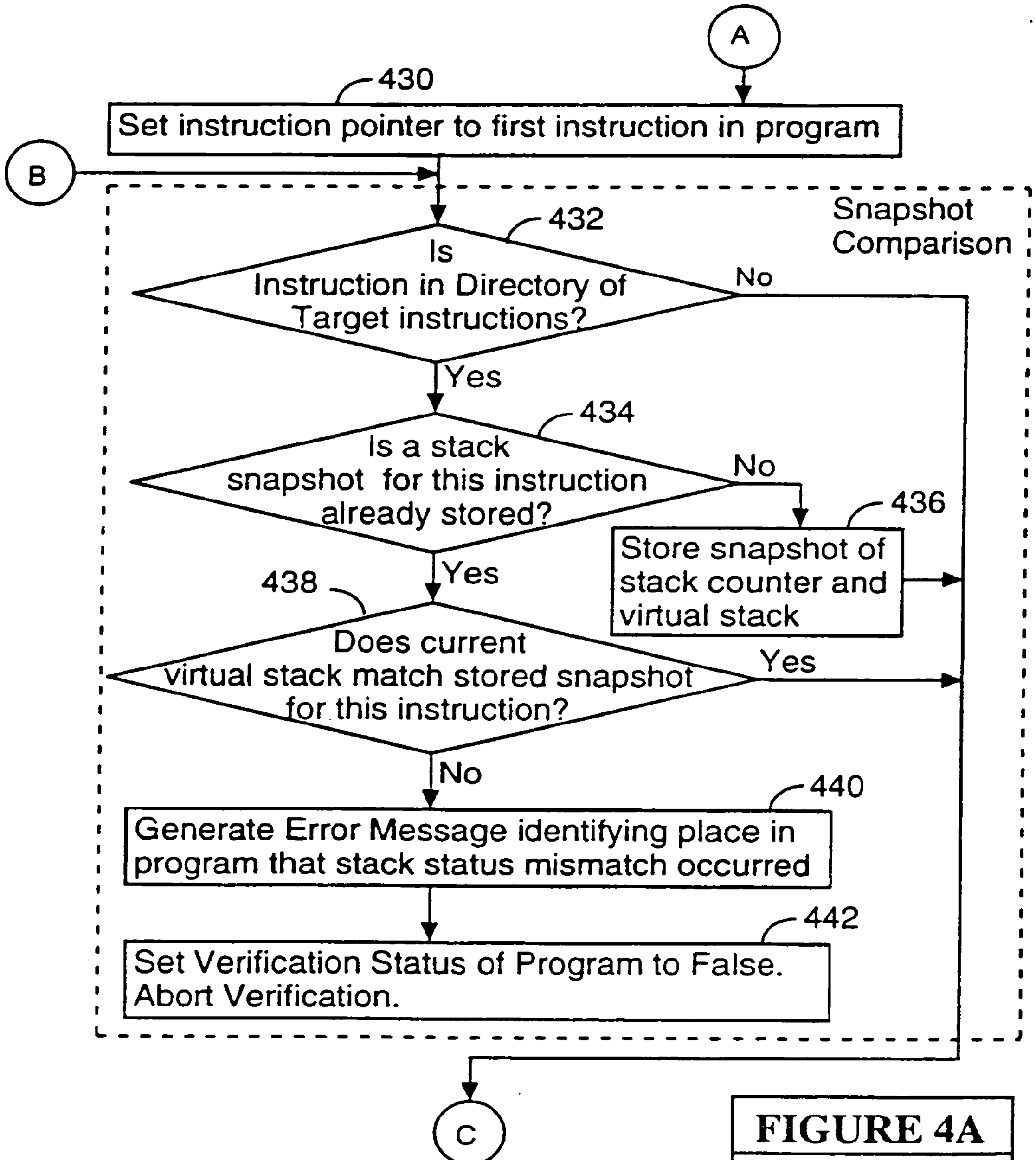


FIGURE 4B

FIGURE 4A
FIGURE 4B
FIGURE 4C
FIGURE 4D
FIGURE 4E
FIGURE 4F
FIGURE 4G

FIGURE 4

BNSDOCID: <EP 0718764A2 I >

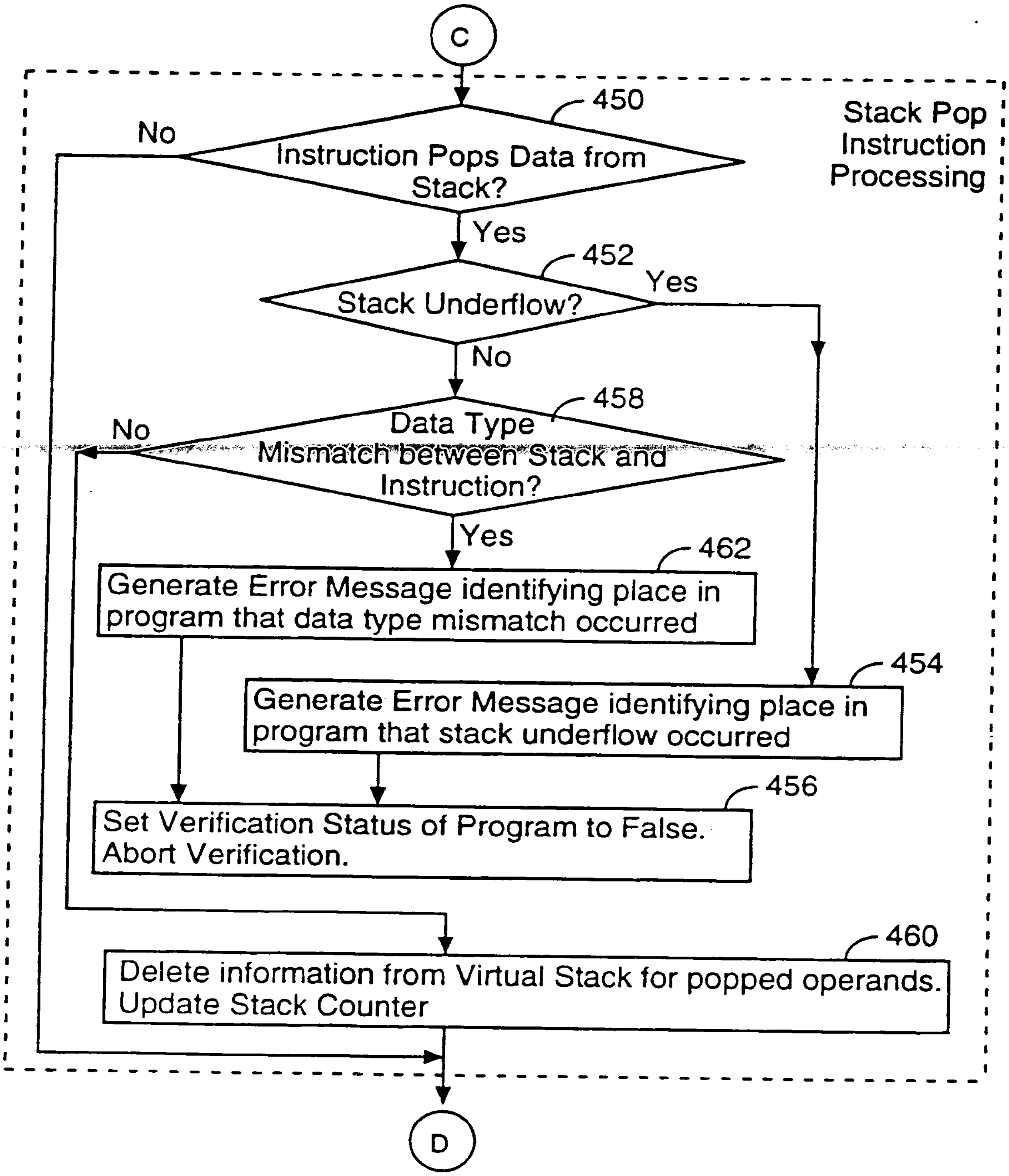


FIGURE 4C

NSDOCID: <EP 0718764A2 I >

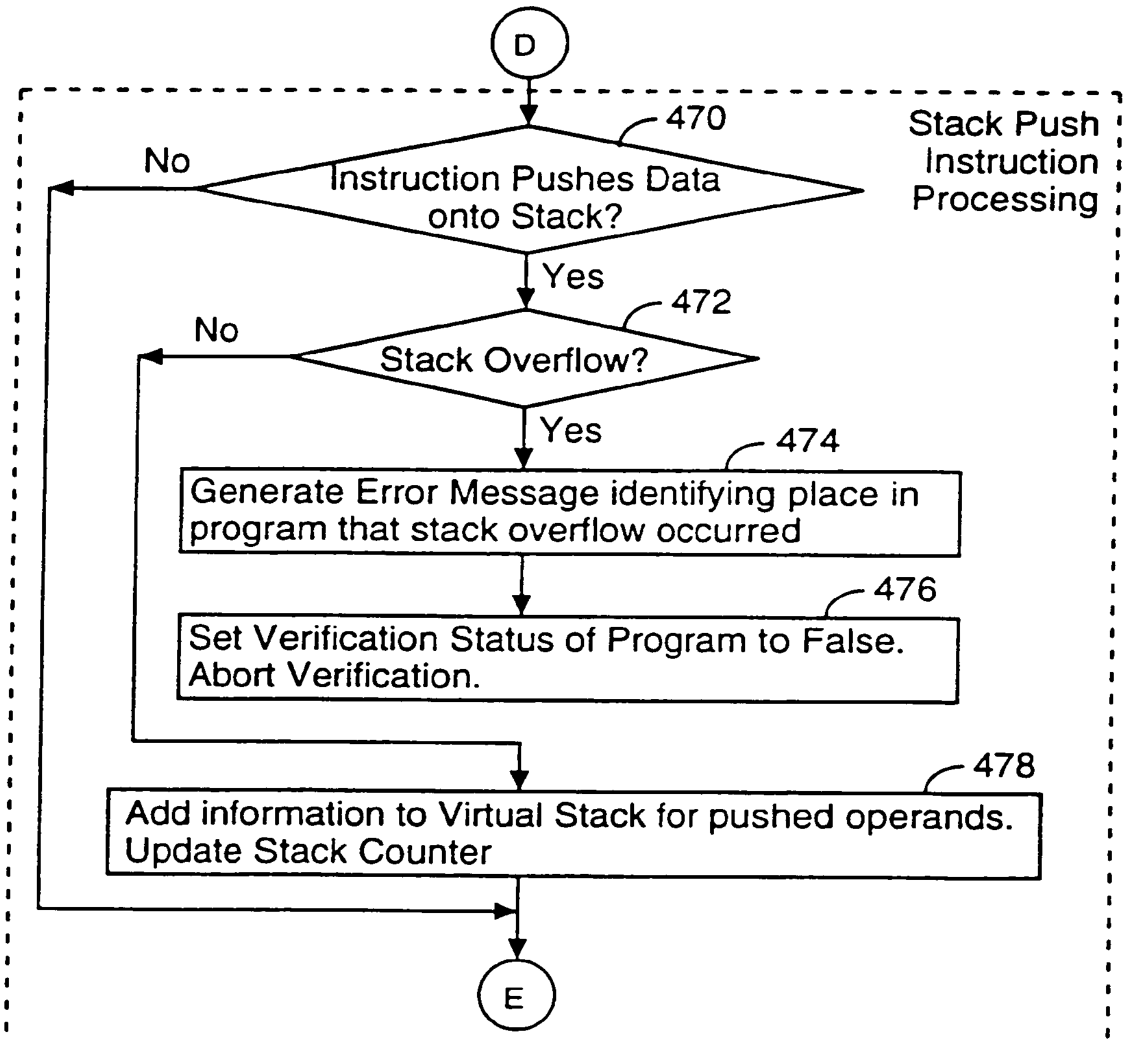


FIGURE 4D

BNSDOCID: <EP 0718764A2 I >

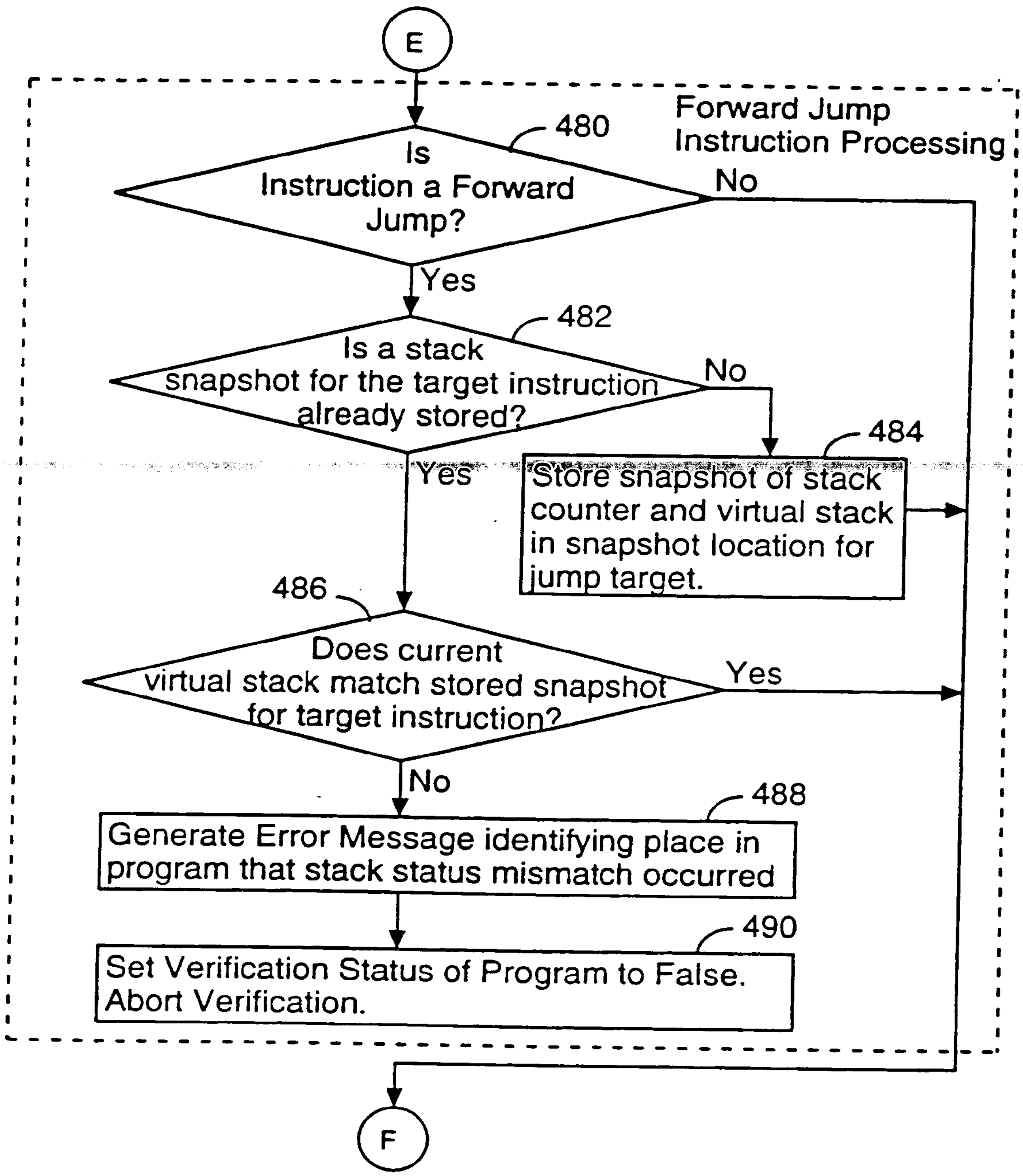


FIGURE 4E

NSDOCID: <EP_ _0718764A2_I_>

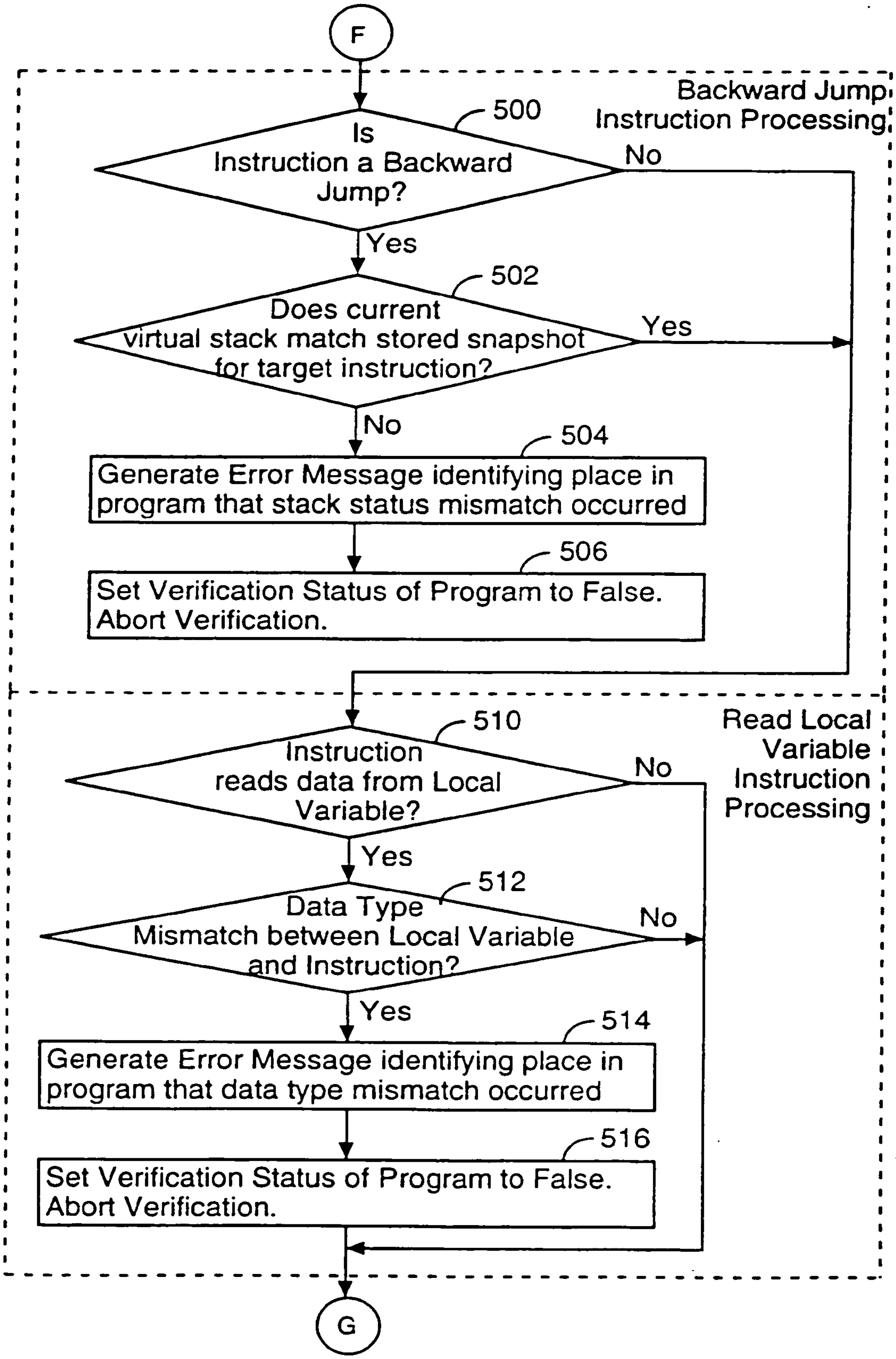


FIGURE 4F

BNSDOCID: <EP 0718764A2 I >

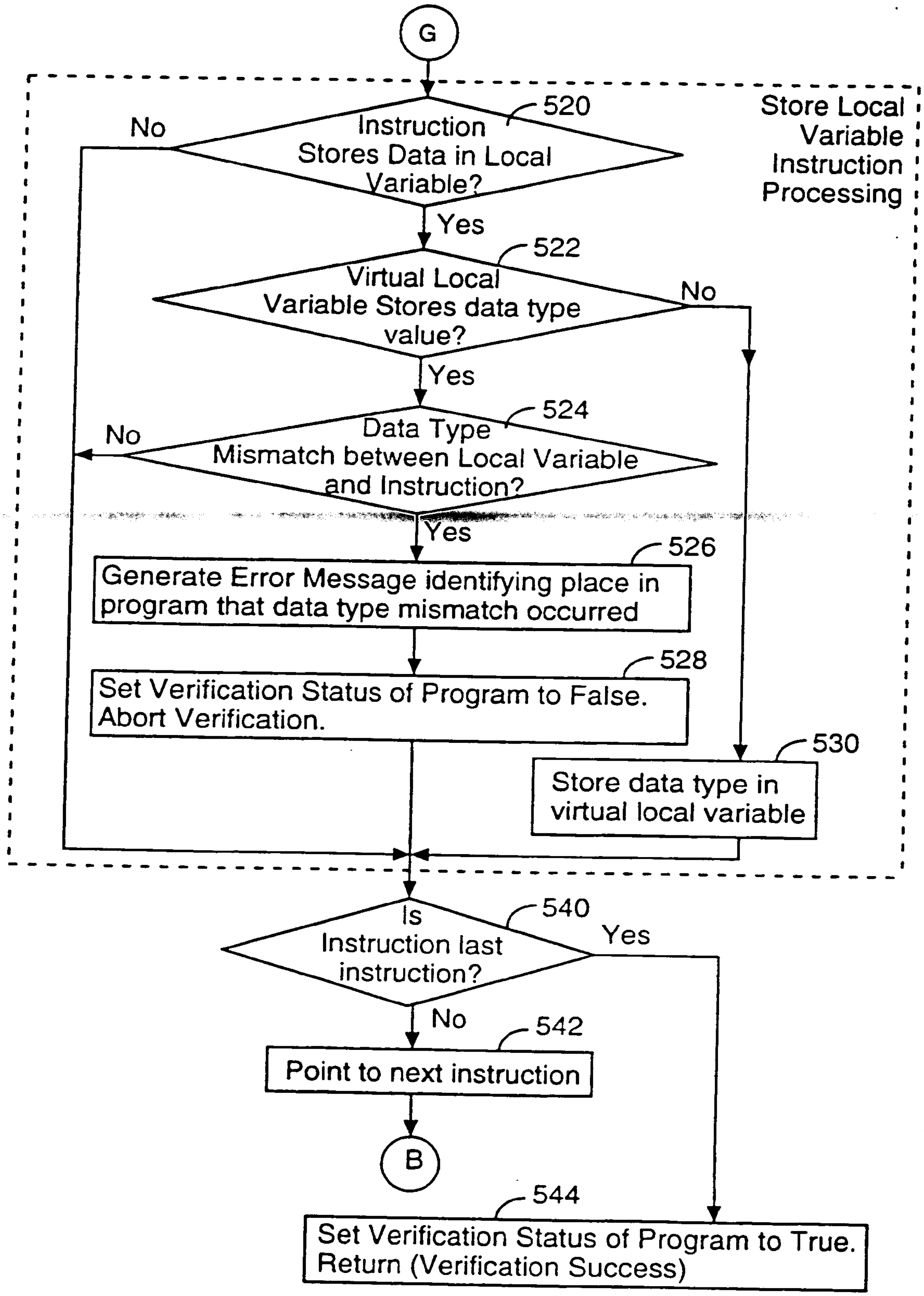


FIGURE 4G

NSDOCID <EP 0718764A2 I >

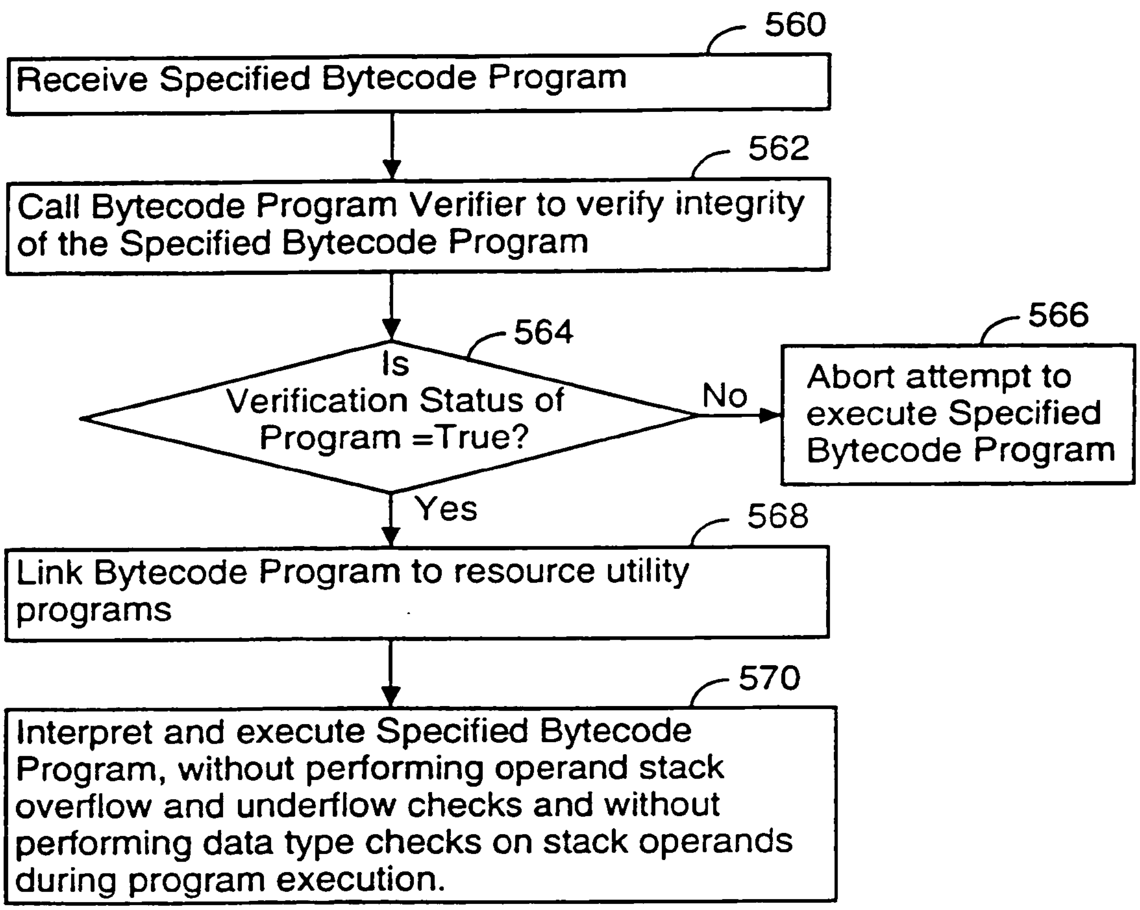


FIGURE 5

BNSDOCID: <EP 0718764A2 I >

This Page Blank (uspto)

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 718 764 A3

(12) EUROPEAN PATENT APPLICATION

(88) Date of publication A3:
15.01.1997 Bulletin 1997/03

(43) Date of publication A2:
26.06.1996 Bulletin 1996/26

(21) Application number: 95120052.6

(22) Date of filing: 19.12.1995

(51) Int. Cl.[6]: **G06F 11/00**, G06F 9/45, G06F 9/44, H04L 29/06

(84) Designated Contracting States:
DE FR GB IT NL SE

(30) Priority: 20.12.1994 US 360202

(71) Applicant: SUN MICROSYSTEMS, INC.
Mountain View, CA 94043 (US)

(72) Inventor: Gosling, James A.
Woodside, California 94062 (US)

(74) Representative: Sparing - Röhl - Henseler
Patentanwälte
Rethelstrasse 123
40237 Düsseldorf (DE)

(54) Bytecode program interpreter apparatus and method with pre-verification of data type restrictions

(57) A program interpreter for computer programs written in a bytecode language, which uses a restricted set of data type specific bytecodes. The interpreter, prior to executing any bytecode program, executes a bytecode program verifier procedure that verifies the integrity of a specified program by identifying any bytecode instruction that would process data of the wrong type for such a bytecode and any bytecode instruction sequences in the specified program that would cause underflow or overflow of the operand stack. If the program verifier finds any instructions that violate predefined stack usage and data type usage restrictions, execution of the program by the interpreter is prevented. After pre-processing of the program by the verifier, if no program faults were found, the interpreter executes the program without performing operand stack overflow and underflow checks and without performing data type checks on operands stored in operand stack. As a result, program execution speed is greatly improved.

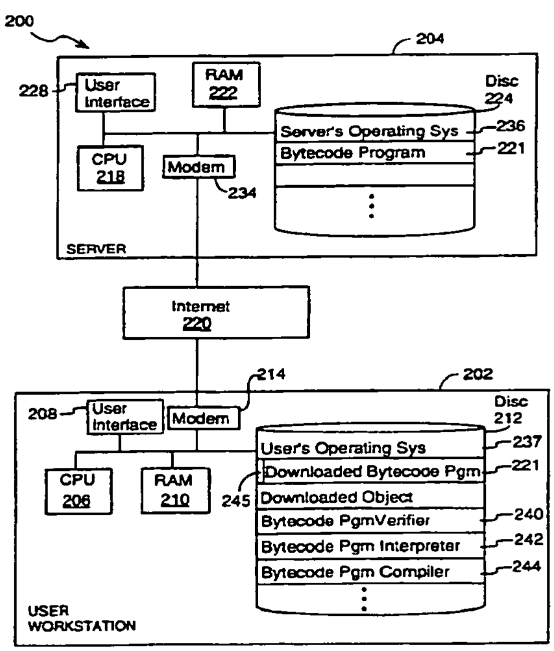


FIGURE 2

EP 0 718 764 A3

BNSDOCID: <EP 0718764A3 I >

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 95 12 0052

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
X Y	SOFTWARE PRACTICE & EXPERIENCE, vol. 17, no. 10, October 1987, CHICHESTER, SUSSEX, GR. BRITAIN, pages 663-683, XP002017341 PERROTT, R.H. AND ADIB ZAREA-ALIABADI: "A Supercomputer Program Development System" * page 664, line 29 - line 32; figure 1 * * page 667, line 29 - line 34 * * page 669, line 13 - line 15 * ---	1,10 2-9, 11-18	G06F11/00 G06F9/45 G06F9/44 H04L29/06
Y	ELEKTRONISCHE RECHENANLAGEN, vol. 24, no. 3, June 1982, MÜNCHEN, DEUTSCHLAND, pages 108-117, XP002017342 SCHAUER, H.: "Architektur und Implementierung eines PASCAL-Systems für Mikrocomputer" * page 116, column 2, line 35 - line 44 * ---	2,5,11, 14	
Y	EP-A-0 424 056 (GE FANUC AUTOMATION NORTH AMER) 24 April 1991 * abstract * * page 8, line 1-4 - line 23-28 * * claims 1-4,8 * ---	3,4,6-9, 12,13, 15-18	TECHNICAL FIELDS SEARCHED (Int.Cl.6) G06F H04L
P,X	ACM SIGPLAN NOTICES, vol. 30, no. 3, 1 March 1995, pages 111-118, XP000567085 GOSLING J: "JAVA INTERMEDIATE BYTECODES ACM SIGPLAN WORKSHOP ON INTERMEDIATE REPRESENTATIONS (IR '95)" * the whole document * --- -/--	1-18	

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	31 October 1996	Fernandez Balseiro,J

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

SDOCID: <EP 0718764A3 I >

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 95 12 0052

DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
A	PROCEEDINGS OF THE CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, ORLANDO, JUNE 27 - 29, 1994, vol. 7 NUMBER 3, 27 June 1994, ASSOCIATION FOR COMPUTING MACHINERY, pages 250-262, XP000522357 WRIGHT A K ET AL: "A PRACTICAL SOFT TYPE SYSTEM FOR SCHEME" * page 250, column 1, line 20 - line 30 * -----	1,2,10, 11	
			TECHNICAL FIELDS SEARCHED (Int.Cl.6)

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	31 October 1996	Fernandez Balseiro,J

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

BNSDOCID <EP 0718764A3 I >

This Page Blank (uspto)